Project no. 033572

CASPAR

Cultural, Artistic and Scientific knowledge for Preservation, Access and Retrieval

Instrument: Information Society Technologies

Thematic Priority: 2.5.10 Access to and preservation of cultural and scientific resources

D2103: Prototype of Descriptive Information-related KM Services

<table>
<thead>
<tr>
<th>Document identifier:</th>
<th>CASPAR-D2103-RP-0101-1_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Submission Date:</td>
<td>18-11-2008</td>
</tr>
<tr>
<td>Due Date:</td>
<td>30-09-2008</td>
</tr>
<tr>
<td>Work package:</td>
<td>2100</td>
</tr>
<tr>
<td>Partners:</td>
<td>All Partners</td>
</tr>
<tr>
<td>WP Lead Partner:</td>
<td>FORTH-ICS</td>
</tr>
<tr>
<td>Document status:</td>
<td>FINAL</td>
</tr>
</tbody>
</table>


Abstract:

This document describes the **Prototype of Descriptive Information-related KM (Knowledge Management) Services**. In particular it describes **SWKM** (Semantic Web Knowledge Middleware) which is one of the key components of the CASPAR system and is being developed by FORTH-ICS. This document provides more details in comparison to earlier deliverables (D2101B).

In addition it describes how these services are already exploited by other components of the CASPAR framework, specifically by the **Finding Aids** component. Actually, one of the engines that is currently used by Finding Aids is SWKM.

As **real users** and data holders requirements should be taken into account, this document also describes the **CNRS DC member tool**. The CNRS DC member tool aims to provide to CASPAR partners a tool which benefits of CASPAR key components features without plunging DC members into the technical aspects of the framework and also enabling them to preserve by access. Thus it bridges the common technical framework of the project with the specific requirements of the DC (Designated Community). It enables people with no KM background to describe the life-cycle of artistic works in a graphical fashion and then export it to the SWKM through a standardised RDF format.

Finally, this document describes a few auxiliary tools, namely **Transformer** and **StarLion**, that have been implemented for aiding data holders to adopt and follow the CASPAR preservation methodology and technology.
**Document Status Sheet**

<table>
<thead>
<tr>
<th>Issue</th>
<th>Date</th>
<th>Comment</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>July 7, 2008</td>
<td>First draft based on the internal deliverable D2101B and on D2102. Extra descriptions for: Main Memory Model Comparison Service</td>
<td>Yannis Tzitzikas, Dimitris Kotzinos, Vassilis Christophides, Yannis Marketakis, Grigoris Antoniou (FORTH-ICS)</td>
</tr>
<tr>
<td>0.2</td>
<td>August 28, 2008</td>
<td>Description of the process of creating instances of ontologies using <strong>Protége</strong> and <strong>Transformer</strong> Added a more detailed description of <strong>Main Memory Model</strong>. Added signature of <strong>Comparison Service</strong>. Added Usage Examples of Client and Services</td>
<td>FORTH-ICS team.</td>
</tr>
<tr>
<td>0.3</td>
<td>September 18, 2008</td>
<td>Various changes and restructurings</td>
<td>FORTH-ICS team.</td>
</tr>
<tr>
<td>0.4</td>
<td>October 16, 2008</td>
<td>CNRS DC member tool</td>
<td>Erik Gebers, Johann Holland and Nicolas Esposito.</td>
</tr>
<tr>
<td>0.5</td>
<td>October 16, 2008</td>
<td>StarLion visualization tool</td>
<td>Yannis Tzitzikas</td>
</tr>
<tr>
<td>0.6</td>
<td>October 17, 2008</td>
<td>Finding Aids instantiated with SWKM</td>
<td>Henri Avancini, Loredana Versenti, Umberto Straccia and Carlo Meghini</td>
</tr>
<tr>
<td>0.7</td>
<td>Nov 1</td>
<td>Changes and improvements all over the document</td>
<td>Yannis Tzitzikas</td>
</tr>
<tr>
<td>Revision</td>
<td>Date</td>
<td>Changes and improvements (in the signatures of services, the examples of Transformer and cross references).</td>
<td>Authors</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
<td>-------------------------------------------------------------------------------------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>0.8</td>
<td>Nov 10</td>
<td>Changes and improvements (in the signatures of services, the examples of Transformer and cross references).</td>
<td>Dimitris Kotzinos, Yannis Tzitzikas, Yannis Marketakis</td>
</tr>
<tr>
<td>0.9</td>
<td>Nov 14</td>
<td>Description of the extension of the CIDOC CRM for digital objects</td>
<td>Martin Doerr, Maria Theodoridou, Yannis Tzitzikas</td>
</tr>
<tr>
<td>0.91</td>
<td>Nov 15</td>
<td>Additional information about the DC member tool</td>
<td>Erik Gebers, Johann Holland and Nicolas Esposito</td>
</tr>
</tbody>
</table>
Prototype of Descriptive Information-related KM Services

Project: CASPAR

Project information

<table>
<thead>
<tr>
<th>Project acronym:</th>
<th>CASPAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project full title:</td>
<td>Cultural, Artistic and Scientific knowledge for Preservation, Access and Retrieval</td>
</tr>
<tr>
<td>Proposal/Contract no.:</td>
<td>IST-2006-033572</td>
</tr>
</tbody>
</table>

Project Officer: Carlos Oliveira

<table>
<thead>
<tr>
<th>Address:</th>
<th>INFSO-E3 Information Society and Media Directorate General Content - Learning and Cultural Heritage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Postal mail:</td>
<td>Bâtiment Jean Monnet (EUFO 1167) Rue Alcide De Gasperi / L-2920 Luxembourg</td>
</tr>
<tr>
<td>Office address:</td>
<td>EUROFORUM Building - EUFO 1167 10, rue Robert Stumper / L-2557 Gasperich / Luxembourg</td>
</tr>
<tr>
<td>Phone:</td>
<td>+352 4301 33052</td>
</tr>
<tr>
<td>Fax:</td>
<td>+352 4301 33190</td>
</tr>
<tr>
<td>Mobile:</td>
<td></td>
</tr>
<tr>
<td>E-mail:</td>
<td><a href="mailto:Carlos.Oliveira@ec.europa.eu">Carlos.Oliveira@ec.europa.eu</a></td>
</tr>
</tbody>
</table>

Project Co-ordinator: David Giaretta

<table>
<thead>
<tr>
<th>Address:</th>
<th>STFC (formerly CCLRC), Rutherford Appleton Laboratory Chilton, Didcot, Oxon OX11 0QX, UK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phone:</td>
<td>+44 1235 446235</td>
</tr>
<tr>
<td>Fax:</td>
<td>+44 1235 446362</td>
</tr>
<tr>
<td>Mobile:</td>
<td>+44 (0) 7770326304</td>
</tr>
<tr>
<td>E-mail:</td>
<td><a href="mailto:d.l.giaretta@rl.ac.uk">d.l.giaretta@rl.ac.uk</a></td>
</tr>
</tbody>
</table>
1. INTRODUCTION .................................................................................................................. 8
  1.1 HOW TO READ THIS DOCUMENT ............................................................................. 9
  1.2 APPLICABLE DOCUMENTS AND REFERENCE DOCUMENTS .............................. 9
  1.3 GLOSSARY .................................................................................................................. 10

2 ARCHITECTURE OF SEMANTIC WEB MODELS .............................................................. 14
  2.1 THE BENEFITS OF ADOPTING SEMANTIC WEB TECHNOLOGIES ...................... 14
  2.2 ABSTRACTING FROM SPECIFIC KNOWLEDGE REPRESENTATION LANGUAGES ............................................. 14
  2.3 ARCHITECTURAL GUIDELINES ................................................................................ 15
  2.4 MODELING GUIDELINES ........................................................................................ 16
  2.5 OASIS AND CIDOC CRM (PROVENANCE AND CONTEXT) ...................................... 19
        2.5.1 Core Ontology (of Intelligibility Dependencies) and CIDOC CRM..................... 19

3 GENERAL ARCHITECTURE OF KNOWLEDGE MANAGEMENT SERVICES ............. 24
  3.1 DESCRIPTIVE METADATA SW MANAGER INTERFACE ......................................... 27
        3.1.1 Browsing Service ............................................................................................. 27

4 BASIC SWKM SERVICES (ANALYTIC DESCRIPTION) .................................................... 32
  4.1 PURPOSE AND SCOPE ............................................................................................ 32
  4.2 BACKGROUND: RDF/S NAMESPACES AND GRAPHSPACES ................................ 32
  4.3 SW KNOWLEDGE REPOSITORY (PERSISTENCE, VALIDATION, QUERY, UPDATE) ................................. 33
  4.4 SW QUERY AND UPDATE LANGUAGES .................................................................. 34
  4.5 BASIC SWKM SERVICES ....................................................................................... 34
  4.6 IMPORT SERVICE .................................................................................................... 34
        4.6.1 Store .................................................................................................................. 35
        4.6.2 Store with Dependencies ................................................................................ 35
  4.7 EXPORT SERVICE ..................................................................................................... 35
        4.7.1 Fetch .................................................................................................................. 35
        4.7.2 Fetch with Dependencies ................................................................................. 36
        4.7.3 Fetch with Data................................................................................................. 36
        4.7.4 Fetch with Data and Dependencies ................................................................. 37
  4.8 QUERY SERVICE ...................................................................................................... 37
  4.9 UPDATE SERVICE ..................................................................................................... 38
  4.10 EXAMPLE OF KNOWLEDGE EVOLUTION ............................................................... 39
  4.11 COMPARISON SERVICES ....................................................................................... 40
  4.12 SWKM MAIN MEMORY MODEL ............................................................................. 43
        4.12.1 About API Access Methods ......................................................................... 45
        4.12.2 More Detailed Description .......................................................................... 45

5 SOFTWARE COMPONENT .................................................................................................. 48
  5.1 ITERATIONS ............................................................................................................. 48
  5.2 MORE DETAILS ON TESTING ............................................................................... 49
  5.3 CASPAR KM INSTALLATION .................................................................................. 52
        5.3.1 Required software components ...................................................................... 52
  5.4 A POSSIBLE DEPLOYMENT ................................................................................... 53
  5.5 SUPPORTED CHARACTERS IN SEMANTIC WEB MIDDLEWARE MANAGER (SWKM) ................................................ 53

6 USING THE SWKM SERVICES: EXAMPLES ................................................................ 55
  6.1 BOOTSTRAPPING .................................................................................................... 55
  6.2 CLIENT’S METHODS ................................................................................................ 56
  6.3 KNOWLEDGE REPOSITORY USAGE EXAMPLE ..................................................... 57
        6.3.1 Importer Example 1 .......................................................................................... 57
        6.3.2 Importer Example 2 .......................................................................................... 57
        6.3.3 Exporter Example 1 .......................................................................................... 58

ISI-2006-033572 PUBLIC
1. INTRODUCTION

This software deliverable is related to Task 2103 Knowledge Manager. The objectives of this task (according to the revised work plan M12-M30) is to design and develop a number high level KM services for digital information preservation systems, all based on Semantic Web Technologies. According to the analysis and the results of the first phase of this project, these services were categorized as follows:

(a) registry-related KM services

These are the services assigned to the component Rep Info Manager in the Conceptual Modeling Deliverable and concern dependency management. The related deliverable has already been delivered (see D2102, M24).

(b) descriptive information-related KM services.

These include services for storing, querying and updating descriptive metadata according to various ontologies. This is the focus of this deliverable.

This document entitled “Prototype of Descriptive Information-related KM (Knowledge Management) Services”, describes SWKM (Semantic Web Knowledge Middleware) which is one of the key components of the CASPAR system and is being developed by FORTH-ICS. This document provides more details in comparison to earlier deliverables (D2101B).

The latest releases of software and documentation for the Semantic Web Knowledge Middleware (basic services, client, main memory model) is continuously available at http://athena.ics.forth.gr:9090/SWKM/

In addition this document describes how these services are already exploited by other components of the CASPAR framework, specifically by the Finding Aids component. Actually, one of the Semantic Engines that is currently used by Finding Aids is SWKM.

The latest version of the software for FINDING AIDS can be downloaded from m http://developers.casparpreserves.eu:8080/hudson/job/CASPAR-FIND

As real users and data holders requirements should be taken into account, this document also describes the CNRS DC member tool. The CNRS DC member tool aims to provide to CASPAR partners a software tool which benefits of CASPAR key components features without plunging DC (Designated Community) members into the technical aspects of the framework and also enabling them to preserve by access. Thus it bridges the common technical framework of the project with the specific requirements of the DC. It enables people with no KM background to describe the life-cycle of artistic works in a graphical fashion and then export it to the SWKM through a standardised RDF format.

The latest releases of software and documentation for CNRS DC member tool can be downloaded from http://www.utc.fr/caspar/wiki/pmwiki.php?n=Main.Proto
Finally, this document describes a few auxiliary tools that have been implemented for aiding data holders to adopt and follow the CASPAR preservation methodology and technology.

This document is important for all designers, developers and users or the CASPAR components. All components are structured according to the CASPAR best practices document and they are accompanied by comments, examples and unit tests.

1.1 HOW TO READ THIS DOCUMENT

This document summarizes the main issues results. More information is available in other deliverables as well as in papers published or to be published in scientific conferences and journals (for more see the Section References). More information about the software is available in the web sites that are mentioned in this document.

1.2 APPLICABLE DOCUMENTS AND REFERENCE DOCUMENTS

Applicable documents

[S0] All previous deliverables of the CASPAR project (especially D2102)


[S3] Y. Tzitzikas, On Preserving the Intelligibility of Digital Objects though Dependency Management, International Conference PV’2007 (Ensuring the Long-Term Preservation and Value Adding to Scientific and Technical Data), Oberpaffenhofen/Munich, Germany, October 2007


Reference documents


1.3 GLOSSARY

[Ax] Applicable Document

[Rx] Reference Document

CASPAR Cultural, Artistic and Scientific knowledge for Preservation, Access and Retrieval

DoW Description of Work

EC European Commission

EPM Executive Project Management

IPC IP Coordinator

IST Information Society Technologies

PACP Partner Administrative Contact Point

PO Project Officer

PPR Project Progress Report

PQE Project Quality Engineer

PTCP Partner Technical Contact Point

R&D Research and Development

SQE Stream Quality Engineer

ST Stream

TN Technical Note

WP Work Package

WPL Work Package Leaders

Designated Community An identified group of potential Consumers who should be able to understand a particular set of information. The Designated Community may be composed of multiple user communities. (OAIS definition)

Archival Information Package (AIP) An Information Package, consisting of the Content Information and the associated Preservation Description Information (PDI), which is preserved within an OAIS. (OAIS definition)
Content Information
The set of information that is the original target of preservation. It is an Information Object comprised of its Content Data Object and its Representation Information. An example of Content Information could be a single table of numbers representing, and understandable as, temperatures, but excluding the documentation that would explain its history and origin, how it relates to other observations, etc. (OAIS definition)

Knowledge Base
A set of information, incorporated by a person or system, that allows that person or system to understand received information. (OAIS definition)

Representation Information
The information that maps a Data Object into more meaningful concepts. An example is the ASCII definition that describes how a sequence of bits (i.e., a Data Object) is mapped into a symbol. (OAIS definition)

Controlled Vocabulary
A controlled vocabulary is a list of terms that have been enumerated explicitly and is controlled by and is available from a controlled vocabulary registration authority. Usually all terms in a controlled vocabulary should have an unambiguous, non-redundant definition.

Glossary
A glossary is a list of terms, usually with textual definitions. The terms may be from a specific subject field or those used in a particular work. The terms are defined within that specific environment and rarely have variant meanings provided. Examples include the EPA Terms of the Environment.

Dictionary
Dictionaries are alphabetical lists of terms and their definitions that provide variant senses for each term, where applicable. They are more general in scope than a glossary. They may also provide information about the origin of the term, variants (both by spelling and morphology), and multiple meanings across disciplines. While a dictionary may also provide synonyms and through the definitions, related terms, there is no explicit hierarchical structure or attempt to group terms by concept.

Gazetteers
A gazetteer is a dictionary of place names. Traditional gazetteers have been published as books or they appear as indexes to atlases. Each entry may also be identified by feature type, such as river, city, or school. Geospatially referenced gazetteers provide coordinates for locating the place on the earth’s surface. An example is the Geographic Names Information Service (http://www-nnd.usgs.gov/www/gnis/). Note that the term gazetteer has several other meanings including an announcement publication such as a patent or legal gazetteer. These gazetteers are often organized using classification schemes or subject categories.

Taxonomy
A taxonomy is a collection of controlled vocabulary terms organized into a hierarchical structure. Each term in a taxonomy is in one or more parent-child relationships to other terms in the taxonomy. There may be different types of parent-child relationships in a taxonomy (e.g., whole-part, genus-species, type-instance), but good practice limits all parent-child relationships to a single parent to be of the same type. Some taxonomies allow a term to have multiple parents.
Thesaurus

A thesaurus is a networked collection of controlled vocabulary terms. This means that a thesaurus uses associative relationships in addition to parent-child relationships. The expressiveness of the associative relationships in a thesaurus vary and can be as simple as "related to term" as in term A is related to term B.

Relationships are generally represented by the notation BT (broader term), NT (narrower term), SY (synonym), and RT (associative or related). Associative Relationships may be more granular in some schemes, e.g. the Unified Medical Language System (UMLS) from the National Library of Medicine has defined over 40 relationships, many of which are associative in nature. Preferred terms for indexing and retrieval are identified. Entry terms (or non-preferred terms) point to the preferred terms that are to be used for each concept. There are standards for the development of monolingual thesauri (NISO, 1998; ISO, 1986) and multi-lingual thesauri (ISO, 1985). However, in these standards the definition of a thesaurus is fairly narrow. Standard relationships are assumed, as well as the identification of preferred terms, and there are specific rules for the creation of the relationships between terms. It should be noted that the definition of a thesaurus in these standards is often at variance with schemes that are actually called thesauri. There are many thesauri that do not follow all the rules of the standard, but are still generally thought of as thesauri. Another type of "thesaurus" represents only equivalence (synonymy), such as the Roget's Thesaurus (with the addition of classification categories).

Many thesauri are very large (more than 50,000 terms) and were developed for a specific discipline, or to support a specific product or family of products. Examples include the Food and Agricultural Organization’s Aquatic Sciences and Fisheries Thesaurus and the NASA Thesaurus for aeronautics and aerospace-related topics.

Other examples: AAT (for arts and architecture) and TGN (for geographic names).

Subject Headings

This scheme provides a set of controlled terms to represent the subjects of items in a collection. Subject heading lists can be extensive, covering a broad range of subjects. However, the subject heading list’s structure is generally very shallow, with a limited hierarchical structure. In use, subject headings tend to be pre-coordinated, with rules for how subject headings can be joined to provide more specific concepts. Examples include the Medical Subject Headings (MeSH) and the Library of Congress Subject Headings (LCSH).
Semantic Networks

With the advent of natural language processing, there have been significant developments in the area of semantic networks. Concepts and terms are structured not as hierarchies but as a network. Concepts are thought of as nodes with various relationships branching out from them. The relationships generally go beyond the standard BT, NT and RT (of thesauri). They may include specific whole-part relationships, cause-effect, parent-child, etc. One of the most noted semantic network is Princeton’s WordNet, which is now used in a variety of search engines.

Ontology

People use the word **ontology** to mean different things, e.g. glossaries & data dictionaries, thesauri & taxonomies, schemas & data models, and formal ontologies & inference. A formal ontology is a controlled vocabulary expressed in an ontology representation language. This language has a grammar for using vocabulary terms to express something meaningful within a specified domain of interest. The grammar contains formal constraints (e.g., specifies what it means to be a well-formed statement, assertion, query, etc.) on how terms in the ontology’s controlled vocabulary can be used together.

People make commitments to use a specific controlled vocabulary or ontology for a domain of interest. Enforcement of an ontology’s grammar may be rigorous or lax. Frequently, the grammar for a "light-weight" ontology is not completely specified, i.e., it has implicit rules that are not explicitly documented.
2 ARCHITECTURE OF SEMANTIC WEB MODELS

2.1 THE BENEFITS OF ADOPTING SEMANTIC WEB TECHNOLOGIES

There are several options for implementing a preservation information system. A promising approach is to adopt Semantic Web technologies. The Semantic Web is generally considered to be the next, revolutionary stage of Web technology and in recent years several applications have been developed in numerous fields. The benefits of adopting Semantic Web technologies has been described in the previous deliverables of the CASPAR project, in brief: (1) expressiveness, namely the ability to encapsulate existing metadata schemata and ontologies, as well as new ones (roughly any “classical” conceptual schema can be represented); (2) formal well-foundedness, derived from the classical object-oriented approach and logics; (3) computational amenability (at least for some dialects of Semantic Web languages); (4) world wide scope and impact. The weaknesses of this approach is that (a) there are no industrial strength platforms (so far), (b) the lack of standardized languages/services for KM (especially for supporting Knowledge Evolution), and (c) scalability (most reasoners are main-memory implementations).

Regarding CASPAR, Semantic Web technologies are be adopted in two places:
1/ for representing the Core Ontology of Intelligibility Dependencies and for enabling the intelligibility-aware processes that are provided by the GapManager (for more see deliverable D2102)
2/ for representing and managing descriptive metadata according to various ontologies.

2.2 ABSTRACTING FROM SPECIFIC KNOWLEDGE REPRESENTATION LANGUAGES

There are already several Semantic Web languages. We should be reluctant in defining ontologies containing sophisticated logic-based modeling constructs. The adoption of logic-based modeling constructs may entails technical risks (e.g. scalability) and there are not technically mature tools for managing this kind of data.

Furthermore, defining classes with necessary and sufficient conditions may also reduce flexibility. For example consider the integrity constraint “a person has only one father”. It may not be appropriate to express explicitly this constraint in a digital preservation environment. For example we may want to attach more than one father to a particular person in case it is not known who of them is actually the father. It is more wise to start modeling assuming only RDFS constructs. RDFS constructs support the standard object-oriented modeling constructs. Since RDFS constructs are also part of OWL-(Lite, DL, Full) this is not going to introduce any interoperability problem. Additional modelling constructs could be added later on and only if it is clear that they are really needed and we are sure that efficient inference mechanisms exist.

A “dual” approach could also be adopted if that is necessary. For instance, an ontology could have 2 versions: one expressed as an RDF Schema and another one that is an OWL DL (or even OWL Full) schema. The first version could be used for the running system (e.g. for making queries). The second can be considered as a “documentation” of the first. For instance the second ontology could include the constraint “each person can have only one father” as this constraint is useful for clarifying how we view the world. The admissible interpretations of this ontology better approximate the possible real world models that correspond to our conceptualization.

In any case CASPAR framework, specifically Finding Aids, could adopt an even more abstract viewpoint. Specifically, it could adopt an additional “parameter” for being able to cope with different languages now or in the future. For instance, we could have a parameter SemWebRepLang whose
values could be RDF, RDFS, OWL Lite, etc. The same could be done for the query and update language, e.g. SemWebQueryLang (with values RQL, SPARQL, etc).

2.3 ARCHITECTURAL GUIDELINES

A preservation information system should be able to manage descriptive metadata. Ontologies and Semantic Web technologies could be exploited for this purpose. Although the forms that descriptive metadata can have, can not be restricted, it would be useful for CASPAR to investigate and propose a general methodology that is based on international standards. For instance, CIDOC CRM is a reference ontology (currently an ISO standard) which could be exploited for this purpose. The CIDOC Conceptual Reference Model (ISO 21127) is a core ontology describing the underlying semantics of data schemata and structures from all museum disciplines and archives and is now being merged with IFLA FRBR concepts. It is result of long-term interdisciplinary work and agreement. It has been derived by integrating (in a bottom-up manner) hundreds of metadata schemas and is stable (almost no change the last 10 years). In essence, it is a generic model of recording of “what has happened” in human scale, i.e. a class of discourse. It can generate huge, meaningful networks of knowledge by a simple abstraction: history as meetings of people, things and information.

FRBRoo is another (under development) ontology that is going to specialize CIDOC CRM. Such ontologies (like CIDOC CRM, FRBRoo) can be exploited for defining domain-specific schemas (in the form of a Semantic Web ontologies). The latter could then be used for describing the objects of interest. The benefit of this approach is that it can alleviate the effort required for defining a domain specific ontology or schema, and that the existence of a general upper level ontology promises interoperability. The partners related to the cultural and the artistic testbed of CASPAR have already found this approach promising and have started working towards this direction (for more information refer to the corresponding deliverables). Figure 2-1 illustrates an architecture of Semantic Web schemas. In particular, CIDOC CRM is modeled as a namespace and FRBRoo as another namespace that specializes it. Under these two we foresee other specializations for capturing domain-specific requirements. For instance, the partners involved in the testbeds could define their schemas there (IRCAM, INA, UofLeeds, UNESCO, ESA, CIANT).

![Figure 2-1 Architecture of Semantic Web Models](image-url)
Over a knowledge repository that contains such schemas, one could define a plethora of high level services. For instance, there is the ubiquitous requirement for Provenance information. A set of provenance-related queries could be designed assuming the CIDOC CRM schema. These queries will return useful information even if applied upon objects which are not described using the CIDOC CRM schema, but a specialization of it. This is an important benefit of adopting semantic technologies and the above architecture of models.

It follows that CIDOC CRM could offer cross-community intelligibility and interoperability for descriptive metadata. Figure 2-2 illustrates the idea.

![Figure 2-2 Cross-Community Interpretability](image)

### 2.4 MODELING GUIDELINES

The purpose of this section is to describe how from an ontology (like CIDOC CRM, FRBRoo) one can define a domain-specific schema (in the form of a Semantic Web ontology) and then use it for documenting the objects of interest. The major part of CIDOC CRM can be straightforwardly represented in Semantic Web languages and such artifacts are already available. However, CIDOC CRM Ontology has nine cases of attributes that start from other attributes (instead of starting from classes). This modeling construct is not supported by Semantic Web languages. However, all these nine attributes aim at capturing type information, therefore they should be expressed as elements in the domain specific schemas. This is clarified by the following example.
Figure 2-3 Links from Links

The left part of Figure 2-3 shows a conceptual diagram illustrating a part of an ontology plus an instantiation of it. In particular, there is a property $\text{ab}$ having domain the class $\text{A}$ and range the class $\text{B}$. There is a property $\text{d}$ whose domain is the property $\text{ab}$, having range the class $\text{C}$. The figure shows also an instantiation of this schema, specifically $\text{a1}$ is an instance of class $\text{A}$, $\text{b1}$ is an instance of class $\text{B}$, and $\text{c1}$ is an instance of class $\text{C}$.

The above logical structure should be implemented as the right side of Figure 2-3. Specifically, we add at our schema (preferable at the domain specific schema) another property named $\text{c1}$ which is defined as subproperty (i.e. specialization) of the $\text{ab}$ property.

The definition of $\text{c1}$ can be placed at the domain specific schema as in Figure 2-4. Actually the right part of Figure 2-4 is a realization of the architecture described in Figure 2-1.
Figure 2-4 Partitioning the knowledge into 3 artifacts

For example, in the “Avis de tempete” instantiation of CRM-CIDOC (that was sketched by IRCAM) we have the P.14.1 property “in_the_role_of”. We can model this by creating a new property named “Composer” as shown in Figure 2-5.

Figure 2-5 Example of Modeling Composer
2.5 OAIS AND CIDOC CRM (PROVENANCE AND CONTEXT)

The PDI of OAIS contains notion of Provenance and Context. CIDOC CRM has adequate concepts to capture provenance and is currently being extended so that to capture the provenance of digital objects too.

The latest version of the extension of CIDOC CRM for digital objects can be downloaded from http://wiki.casparpreserves.eu/pub/Main/DemoData/CIDOC_Digital.rdf

Its representation in RDF is in order:

```xml
  <rdfs:Class rdf:ID="C1_Digital_Object">
    <rdfs:comment></rdfs:comment>
    <rdfs:subClassOf rdf:resource="http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf#E54_Dimension" />
    <rdfs:subClassOf rdf:resource="http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf#E73_Information_Object" />
  </rdfs:Class>

  <rdfs:Class rdf:ID="C2_Digitization_Process">
    <rdfs:comment></rdfs:comment>
    <rdfs:subClassOf rdf:resource="http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf#E11Modification" />
    <rdfs:subClassOf rdf:resource="http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf#E16Measurement" />
    <rdfs:subClassOf rdf:resource="http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf#E65Creation" />
  </rdfs:Class>

  <rdfs:Class rdf:ID="C3_Formal_Derivation">
    <rdfs:comment></rdfs:comment>
    <rdfs:subClassOf rdf:resource="http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf#E65Creation" />
  </rdfs:Class>

  <rdfs:Class rdf:ID="C4_Norm">
    <rdfs:comment></rdfs:comment>
  </rdfs:Class>

  <rdfs:Class rdf:ID="C5_Copyright">
    <rdfs:comment></rdfs:comment>
    <rdfs:subClassOf rdf:resource="http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf#E30Right" />
  </rdfs:Class>

  <rdfs:Class rdf:ID="C6_Copying">
    <rdfs:comment></rdfs:comment>
    <rdfs:subClassOf rdf:resource="http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf#E7Activity" />
  </rdfs:Class>

  <rdfs:Property rdf:ID="S1F_digitized">
    <rdfs:domain rdf:resource="#C2_Digitization_Process" />
    <rdfs:range rdf:resource="http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf#E18Physical_Thing" />
  </rdfs:Property>

  <rdfs:Property rdf:ID="S1A_was_digitized_by">
    <rdfs:domain rdf:resource="http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf#E18Physical_Thing" />
    <rdfs:range rdf:resource="#C2_Digitization_Process" />
  </rdfs:Property>

  <rdfs:Property rdf:ID="S2F_used_as_source">
    <rdfs:domain rdf:resource="#C3_Formal_Derivation" />
    <rdfs:range rdf:resource="http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf#E70Thing" />
  </rdfs:Property>

  <rdfs:Property rdf:ID="S2A_was_source_for">
    <rdfs:domain rdf:resource="http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf#E70Thing" />
    <rdfs:range rdf:resource="#C3_Formal_Derivation" />
  </rdfs:Property>

  <rdfs:Property rdf:ID="S3E_allows">
    <rdfs:domain rdf:resource="http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf#E30Right" />
  </rdfs:Property>
</rdf:RDF>
```
The following figures depict examples of modeling provenance using CIDOC CRM.
Figure 6 Modeling the Change of Custody Using CIDOC CRM

Figure 7 Modeling the Derivation History Using CIDOC CRM

The adoption of the extension of CIDOC CRM in conjunction with the query service described in Section 4.8 and Section 6.3.5 enables a plethora of provenance-related access and query services in a straightforward manner.
Context is very hard to define. By definition information has a value only if placed at context. To this end OAIS does not provide any help and it is rather weak. The application of CIDOC CRM in a universal scope could be a solution to this problem.

2.5.1 Core Ontology (of Intelligibility Dependencies) and CIDOC CRM

One rising question is how the core ontology of Gap Manager (see D2102) could be related with other ontologies of wider scope. Below we discuss in brief the case of CIDOC CRM.

The class Module may be mapped to the entity E73 Information Object or with C1 Digital Object. The latter is a newly added entity of CIDOC CRM, defined as a subclass of E73 Information Object and of E54 Dimension. The corresponding part of the CIDOC CRM ontology is shown next.
The relation \textit{depends} (between Modules) may be mapped to the property \textbf{P2 has Type}. An example is given in the next figure that shows 3 images (Crete.jpg, Crete.png, CreteSmall.png) all defined as instances of the class \textbf{C1 Digital Object}. Notice the use of the \textbf{P2 has type} relation.

In the general case, one may find out that a intelligibility dependency corresponds to certain relationships, or paths of relationships, over an existing conceptual model. Then he could either "merge appropriately" the schema of his conceptual model with that of the Gap Manager. For instance if he wants every CIDOC CRM E73\_Information\_Object to be considered as Module, then he could define E73\_Information\_Object as a subclass of Module. Alternatively, we could either manually or through a declarative update language, classify his data also wrt the gap mgr ontology.

In that case the gap manager ontology could be considered as the schema of a read-only view of more sophisticated conceptual models. The ability of multiple-classification and of inheritance (of Semantic Web languages) gives this flexibility.
3 GENERAL ARCHITECTURE OF KNOWLEDGE MANAGEMENT SERVICES

A layered KM architecture is more appropriate for this project (and for digital preservation in general). At the lower layer, a general purpose Semantic Web Knowledge Middleware (SWKM) provides a set of core services for managing Semantic Web data. At the upper layer, a CASPAR Knowledge Manager (CKM) provides high level services based on the OAI-PMH model (and its extensions) aiming at offering an abstraction useful for preservation information systems. CKM is implemented using services offered by SWKM. The specification of CKM should be generic and less probable to change over time. The lower layer could change in future, or different implementations for it may be available (now or in the future).

This architecture is used for implementing the intelligibility-related services and the general descriptive metadata services. Below we discuss the latter case.

Consider the case where formally expressed knowledge is available (in the form of ontologies and instantiations). One approach would be to keep these models stored in files (e.g. RDF expressed in XML). However, a more advanced approach is to allow querying these models. In such cases a SW repository is proposed. Moreover this approach allows updating the expressed knowledge using declarative languages. However, these approaches are not mutually exclusive.
The description of the components follows

<table>
<thead>
<tr>
<th>Component</th>
<th>Knowledge Manager</th>
</tr>
</thead>
</table>
| Responsibilities | Capture Higher level Semantics  
| | Manage Designated Community Knowledge profile  
| | Identify ReplInfo Gaps  
| | Manage Ontologies and Metadata |
| Parts | It comprises two layers  
| | A) SWKM (Semantic Web Knowledge Manager) which is the lower layer  
| | B) CKM (CASPAR Knowledge Manager) which is the upper layer  
| | SWKM (Semantic Web Knowledge Manager) will provide a set of core services for managing Semantic Web data.  
| | At the upper layer, CKM will provide high level services based on the OAIS model aiming at offering an abstraction useful for preservation information systems. CKM can be implemented using services offered by SWKM. |
| Provided Interfaces | (A) CKM  
| | DCPProfileManager  
| | ReplInfoGapManager  
| | DescriptiveMetadataSWManager  
| | (B) SWKM  
| | Services:  
| | Query ()  
| | Update ()  
| | Import()  
| | Export()  
| | SWMainMemoryManagement  
| | SWMainMemoryModel |

CKM aims at enabling the intelligibility-aware services that were described in D2101B. Specifically it provides the interfaces DCPProfileManager and ReplInfoGapManager (which are described in D2101B), and the interface DescriptiveMetadataSWManager (which is described in more detail in a later Section).

<table>
<thead>
<tr>
<th>Interface</th>
<th>DescriptiveMetadataSWManager</th>
</tr>
</thead>
</table>
| Operations | getDescriptiveMetadata(object)  
| | getDescriptiveMetadata(object, ontology)  
| Used by | Finding Aids |
The low level (knowledge repository) basic services provides scalable persistence services for large volumes of ontologies and ontology-based descriptions. Access and manipulation are supported by declarative **query** and **update** SW languages. Compared to API-based knowledge application development, the languages supported by the Knowledge Repository aim at satisfying the requirements of CASPAR KM services applications for expressiveness (the ability to express accurately what the query/update initiator wants), generality (the ability to implement easily new query/update functionality) and performances (the ability to respond in a fast way to a query/update request). Following the principle of simplicity and generality only a small number of basic services will be considered. This is for hiding the internal implementation details for making it easier to re-implement these services in the future.

<table>
<thead>
<tr>
<th><strong>Interface</strong></th>
<th>SWKM Services</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operations</strong></td>
<td>query</td>
</tr>
<tr>
<td></td>
<td>update</td>
</tr>
<tr>
<td></td>
<td>import</td>
</tr>
<tr>
<td></td>
<td>export</td>
</tr>
<tr>
<td><strong>Used by</strong></td>
<td>Data Access Manager and Security</td>
</tr>
<tr>
<td></td>
<td>Finding Aids</td>
</tr>
</tbody>
</table>
3.1 DESCRIPTIVE METADATA SW MANAGER INTERFACE

In general the Descriptive Metadata Services are the Basic SW KM Services that are described in more detail in Section X.

However, some more abstract services have been requested. For instance the Virtualization component has requested the following service:

<table>
<thead>
<tr>
<th>Service</th>
<th>Requested By</th>
</tr>
</thead>
<tbody>
<tr>
<td>getRelatedConcepts(Concept, RelationshipType): Concepts[]</td>
<td>Virtualization</td>
</tr>
</tbody>
</table>

The parameter Concept will be a concept identified by its URI.

The parameter RelationshipType could be one of the following:

- DirectSubClasses
- AllSubClasses
- SuperClasses
- AllSuperClasses

The response of this method (which is a set of URIs) will be based on the semantics of subClassOf relation.

In additions Finding Aids have requested the following:

<table>
<thead>
<tr>
<th>Methods</th>
<th>Requested From</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDescriptiveMetadata(object)</td>
<td>FindingAids</td>
<td></td>
</tr>
</tbody>
</table>

More detailed analysis:

- Find the description of a resource with a given URI
- Find the description of resources excluding descriptions related to a particular class
- Find the classes under which a resource has been classified

getDescriptiveMetadata(object, ontology) FindingAids

More detailed description:

- Get the classes of resource that belong to particular ontology

All of the above can be provided by a straightforward application of the query service. However a new browsing service has been designed.

3.1.1 Browsing Service
This service allows navigating the graphs defined by the ontologies and resource descriptions that are stored in SWKM repository. The signature of this service is:

```
get(String conceptUri, ServiceType sType, ConceptType cType) : List<String[]>
```

The first parameter, `conceptUri`, is either the string representation of a URI that corresponds to an existing resource or a string pattern.

The second parameter, i.e. `sType`, is of type `ServiceType` which is an enumerated type with the following values:
- `CLASSES`
- `INSTANCES`
- `ALLSUB`
- `DIRSUB`
- `ALLSUP`
- `DIRSUP`
- `FROM`
- `TO`
- `PATTERN_MATCH`

The third parameter, i.e. `cType`, is used for specifying the type of the first parameter (conceptURI). The type `ConceptType` is an enumerated type with the following values:
- `CLASS`
- `PROPERTY_CLASS`
- `CLASS_INSTANCE`

This service returns a list of String[] that may represent URIs and Literals. The following table shows the meaningful combinations of `sType` and `cType` parameters, and what this services returns as response.

<table>
<thead>
<tr>
<th>ServiceType</th>
<th>conceptType</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASSES</td>
<td>CLASS</td>
<td>The metaclass URIs under which the given class is classified</td>
</tr>
<tr>
<td></td>
<td>PROPERTY_CLASS</td>
<td>The metaclass URI under which the given property is classified</td>
</tr>
<tr>
<td></td>
<td>CLASS_INSTANCE</td>
<td>The class URIs under which the specified instance is classified</td>
</tr>
<tr>
<td>INSTANCES</td>
<td>CLASS</td>
<td>All Instances of the given Class URI</td>
</tr>
<tr>
<td></td>
<td>PROPERTY_CLASS</td>
<td>All Instances of the given Property URI</td>
</tr>
<tr>
<td></td>
<td>CLASS_INSTANCE</td>
<td>nothing</td>
</tr>
<tr>
<td>ALLSUB</td>
<td>CLASS</td>
<td>All subClass URIs of the given class URI</td>
</tr>
<tr>
<td></td>
<td>PROPERTY_CLASS</td>
<td>All subProperty URIs of the given property URI</td>
</tr>
<tr>
<td></td>
<td>CLASS_INSTANCE</td>
<td>nothing</td>
</tr>
<tr>
<td>Requirement</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>DIRSUB</td>
<td>CLASS: The direct superClass URIs of the given property URI. PROPERTY_CLASS: The direct subProperty URIs of the given property URI. CLASS_INSTANCE: nothing.</td>
<td></td>
</tr>
<tr>
<td>ALLSUP</td>
<td>CLASS: All superClass URIs of the given property URI. PROPERTY_CLASS: All superProperty URIs of the given property URI. CLASS_INSTANCE: nothing.</td>
<td></td>
</tr>
<tr>
<td>DIRSUP</td>
<td>CLASS: The direct superClass URIs of the given property URI. PROPERTY_CLASS: The direct superProperty URIs of the given property URI. CLASS_INSTANCE: nothing.</td>
<td></td>
</tr>
<tr>
<td>FROM</td>
<td>CLASS: All property URIs that have the specified Class as domain. PROPERTY_CLASS: The domain of the specified Property. CLASS_INSTANCE: nothing.</td>
<td></td>
</tr>
<tr>
<td>TO</td>
<td>CLASS: All property URIs that have the specified class as range. PROPERTY_CLASS: The range of the specified Property. CLASS_INSTANCE: nothing.</td>
<td></td>
</tr>
<tr>
<td>PATTERN_MATCH</td>
<td>CLASS: The Class URIs that match the given pattern. PROPERTY_CLASS: The Property URIs that match the given pattern. CLASS_INSTANCE: The instance URIs that match the given pattern.</td>
<td></td>
</tr>
</tbody>
</table>

The results of the service, as already mentioned, are meaningful in combination with the specified parameters. Additionally the results may be a single URI (i.e. when ServiceType=FROM & ConceptType=PROPERTY_CLASS), a list of URIs (i.e. when ServiceType=INSTANCE & ConceptType=CLASS) or a pair of URIs (i.e. when ServiceType=INSTANCE & ConceptType=PROPERTY_CLASS). This led to the result of a list of String[].

If the given conceptURI does not exist or there is nothing to return then an empty array is returned (particularly it is returned a list with an empty array in it). In addition if the specified URI (conceptURI) is not valid then the result is an empty array again.
Below are some examples of the service and the results with some combinations of the parameters.

- get("http://localhost:3030/schema#Student",ServiceType.CLASSES,ConceptType.CLASS)
  - "http://www.w3.org/2000/01/rdf-schema#Class"
- get("http://localhost:3030/schema#assists",ServiceType.CLASSES,ConceptType.PROPERTY_CLASS)
  - "http://www.w3.org/1999/02/22-rdf-syntax-ns#Property"
- get("http://localhost:3030/data/John",ServiceType.CLASSES,ConceptType.CLASS_INSTANCE)
  - []
- get("http://localhost:3030/schema#Student",ServiceType.INSTANCES,ConceptType.CLASS)
  - "http://localhost:3030/data/John"
  - "http://localhost:3030/data/Mark"
- get("http://localhost:3030/schema#attends",ServiceType.INSTANCES,ConceptType.PROPERTY_CLASS)
- get("http://localhost:3030/data/John",ServiceType.INSTANCES,ConceptType.CLASS_INSTANCE)
  - []
- get("http://localhost:3030/schema#Person",ServiceType.ALLSUB,ConceptType.CLASS)
  - "http://localhost:3030/schema#Student"
• get("http://localhost:3030/schema#registers",ServiceType.ALLSUB,ConceptType.PROPERTY_CLASS)
  o “http://localhost:3030/schema#joins"
  o “http://localhost:3030/schema#attends"
  o “http://localhost:3030/schema#assists"

• get("http://localhost:3030/data/Java",ServiceType.ALLSUB,ConceptType.CLASS_INSTANCE)
  o [ ]

• get("http://localhost:3030/schema#Person",ServiceType.DIRSUB,ConceptType.CLASS)
  o “http://localhost:3030/schema#Student"

• get("http://localhost:3030/schema#registers",ServiceType.DIRSUB,ConceptType.PROPERTY_CLASS)
  o “http://localhost:3030/schema#joins"
  o “http://localhost:3030/schema#students”

• get("http://localhost:3030/data/Mark",ServiceType.DIRSUB,ConceptType.CLASS_INSTANCE)
  o [ ]

• get("http://localhost:3030/schema#Undergraduate_Student",ServiceType.ALLSUP,ConceptType.CLASS)
  o “http://localhost:3030/schema#Student”
  o “http://localhost:3030/schema#Person”

• get("http://localhost:3030/schema#assists",ServiceType.ALLSUP,ConceptType.PROPERTY_CLASS)
  o “http://localhost:3030/schema#joins"
  o “http://localhost:3030/schema#registers”

• get("http://localhost:3030/data/Mark",ServiceType.ALLSUP,ConceptType.CLASS_INSTANCE)
  o [ ]

• get("http://localhost:3030/schema#Undergraduate_Student",ServiceType.DIRSUP,ConceptType.CLASS)
  o “http://localhost:3030/schema#Student”

• get("http://localhost:3030/schema#assists",ServiceType.DIRSUP,ConceptType.PROPERTY_CLASS)
  o “http://localhost:3030/schema#joins"

• get("http://localhost:3030/data/Mark",ServiceType.DIRSUP,ConceptType.CLASS_INSTANCE)
  o [ ]

• get("http://localhost:3030/schema# Student",ServiceType.FROM,ConceptType.CLASS)
  o “http://localhost:3030/schema#registers”
  o “http://localhost:3030/schema#joins"

• get("http://localhost:3030/schema#assists",ServiceType.FROM,ConceptType.PROPERTY_CLASS)
  o “http://localhost:3030/schema#Postgraduate_Student”

• get("http://localhost:3030/data/Java",ServiceType.FROM,ConceptType.CLASS_INSTANCE)
  o [ ]

• get("http://localhost:3030/schema# Student",ServiceType.TO,ConceptType.CLASS)
  o [ ]

• get("http://localhost:3030/schema#assists",ServiceType.TO,ConceptType.PROPERTY_CLASS)
  o “http://localhost:3030/schema#Lesson”

• get("http://localhost:3030/data/Java",ServiceType.TO,ConceptType.CLASS_INSTANCE)
4 BASIC SWKM SERVICES (ANALYTIC DESCRIPTION)

4.1 PURPOSE AND SCOPE

Just an introduction to the key concepts. More information is available at the prototype description.

4.2 BACKGROUND: RDF/S NAMESPACEs AND GRAPHSPACES

The knowledge models of CASPAR (see glossary at the appendix) can be represented as RDF schemas. The preservation metadata can be represented as RDF descriptions which will comply to RDF schemas. To this end, the knowledge repository will be able to store, retrieve and update RDF/S schemas and descriptions based on the name or graph spaces they belong:

Namespace (or named schemas): a collection of RDFS class or property names (i.e. graph labels), identified by a URI reference, which can be employed in the description of a digital object. Given that these names are unambiguously defined in an RDFS schema, classes and properties are universally qualified by their name, which is prefixed by the URI of the namespace (i.e. the schema) they belong. Thus, namespaces provide a standard way of distinguishing among classes and properties that carry the same names in different schemas regardless whether their meaning is the same or not.

Graphspace (or named graphs): a collection of RDF triples (i.e. graph edges), identified by a URI reference, that form the description of a digital object. Given that these URIs essentially identify RDF/S (sub-)graphs, they can be also employed as source or target resources of properties, so forming complex RDF/S hyper-graphs (i.e. graphs whose nodes are graphs). There are no constraints regarding the contents (disjointness or containment) of a graphspace: the same (subset of) triples may belong to different graphspaces while graphspaces may be composed of both schema and data triples. Thus, graphspaces provide a standard way for distinguishing among descriptions provided by different actors (or sources), for restricting information access and supporting access control, versioning etc.

Name or graph spaces will be the core mechanism for abstracting from the syntax and semantic peculiarities of the RDF/S data model. Their unique URIs unambiguously identifies name or graph spaces produced and consumed in a CASPAR application space.

[ ]

- get("Student",ServiceType.PATTERN_MATCH,ConceptType.CLASS)
  - http://localhost:3030/schema#Student
  - http://localhost:3030/schema#Undergraduate_Student
  - http://localhost:3030/schema#Postgraduate_Student
- get("it",ServiceType.PATTERN_MATCH,ConceptType.CLASSPROPERTY)
  - http://localhost:3030/schema#attends
- get("a",ServiceType.PATTERN_MATCH,ConceptType.CLASSINSTANCE)
  - http://localhost:3030/data/Mark
  - http://localhost:3030/data/Java
  - http://localhost:3030/data/Databases
4.3 SW KNOWLEDGE REPOSITORY (PERSISTENCE, VALIDATION, QUERY, UPDATE)

Several RDF stores have been developed during the last four years for supporting real-scale Semantic Web applications. They usually rely on (main-memory) virtual machine-implementations or on (object-) relational database technology while exploit a variety of storage schemes. The implementation of the CASPAR KM will rely on RDFSuite. We have implemented an appropriate APIs for bulk loading and exporting of both resource descriptions and schemata (given a name or graph space) in file streams (RDF/XML or triple-based ascii formats).

4.4 SW QUERY AND UPDATE LANGUAGES

Several query languages (e.g. RQL: http://139.91.183.30:9090/RDF/RQL/, SPARQL: http://www.w3.org/TR/rdf-sparql-query/) have been developed during the last five years for supporting declarative access to ontologies and resources descriptions available on the Semantic Web. One of the unique features of RQL is its ability to match filtering/navigation patterns against RDF/S graphs by taking into account (or ignoring) the semantics (e.g. transitivity of subsumption relationships) of the ontologies employed to describe knowledge artifacts (see [Haase et al 2004] for a detailed comparison of SW QL expressiveness). This functionality is useful for abstracting the technicalities of the RDF/S data model from the end-user CASPAR applications while it has been efficiently implemented in secondary memory [Christophides et al 2003]. For these reasons we rely on the RDFSuite RQL implementation, to support the CASPAR KM query service. We are currently extending RQL filtering/navigation patterns to support graphspaces (namespaces are already supported). Furthermore, there is ongoing implementation of the first declarative language for inserting/deleting/modifying arbitrary RDF/S (or fragments of OWL) (sub)graphs. The language, called RUL [Magiridou et al 2005], ensures that the execution of the update primitives on nodes and arcs neither violates the semantics of the RDF model (e.g. insert a property as an instance of a class) nor the semantics of a specific RDFS schema (e.g. modify the subject of a property with a resource not classified under its domain class). This main design choice has been made given that type safety for updates is even more important than type safety for queries: the more errors we can catch at update specification time the less costly runtime checks (and possibly expensive rollbacks) we need. The rest of the design choices concern (a) the granularity of the supported update primitives; (b) the deterministic or not behavior of the executed sequences of update statements; (c) the smooth integration with an underlying RDF/S query language like RQL. For these reasons we rely on the RDFSuite RUL implementation, to support the CASPAR KM update service.

4.5 BASIC SWKM SERVICES

It is important to note that the Basic SWKM Services have been designed with intelligibility in mind. This explains why the basic (import and export) services are offered in two “modes”: one plain and another one that is intelligibility-aware, i.e. dependency-aware.

For instance, the Import service (specifically storeWithDependencies) is intelligibility-aware as it tries to “fill the gap” (so that to ensure that the repository contains only intelligible knowledge artifacts). Specifically it identifies the dependencies of the SW files to be imported and tries to fill the intelligibility gap by downloading the needed files from the network.

Analogously, the Export service (specifically fetchWithDependencies) is intelligibility-aware as it tries to return self-intelligible packages. An extension could be to add to the storeWithDependencies an additional parameter holding those URIs that the requester is supposed to know (have) already. However the dependency relationships are not expected to be very long so this extension should not have high priority.

Each of the services is described below in more details.
4.6 IMPORT SERVICE

The Import Service is responsible for loading the contents of a (set of) valid and well-formed namespace(s) or named graph(s) to the underlying persistent storage medium. It is also responsible for creating the necessary database constructs (tables, relationships, indices) that allow for efficient retrieval and manipulation by the RQL/RUL interpreter. The service uses the main memory representation described earlier. It firstly loads the RDF content into the main memory, checks validation constraints on it and, if it is deemed valid, commits it to the repository. The RDF model is afterwards unloaded from the main memory.

4.6.1 Store

Signature

```
public void store(
    java.lang.String[] uris,
    java.lang.String[] data,
    java.lang.String formatDesc,
    gr.forth.ics.rdfsuite.services.DbSettings dbSettings)
```

Description

The `store` method is responsible for storing the contents of a (set of) valid and well-formed namespace(s) or named graph(s) currently in memory to the underlying persistent storage medium. It is also responsible for updating the necessary database constructs (tables, relationships, indices) that allow for efficient retrieval and manipulation by the RQL/RUL interpreter.

Preconditions

- The array of URIs has exactly the same number of elements as the array of data.
- Format is either TRIG or RDF/XML.
- If a namespace or named graph declares its own URI (in a syntax-specific way), it must be the same as the respective entry in the array of URIs.

Effects

After the successful execution of the operation, the underlying storage will contain all supplied RDF namespaces and named graphs, identified by their respective URIs. During storage, all redundant information in the original data is removed.

Exceptions Thrown

The following exceptions are thrown by the service:

```java
void store(String[] URIs, String[] documents, String format)
void storeWithDependencies(String[] URIs, String[] documents, String format)
```
• **LabelCreationException**: if something went wrong in the label creation.
• **NamespaceStoringException**: if a namespace already stored in the db is attempted to be stored again.
• **ValidationException**: if the schema or data are not valid.
• **ParsingException**: in case the parsing of a document fails.
• **RdfSuiteException**: for any other unexpected failure.
• **NamespaceUriFormatException**: this exception is thrown when a given namespace URI does not have a correct format (must end with #).

### 4.6.2 Store with Dependencies

**Signature**

```java
public void storeWithDependencies(
    java.lang.String[] uris,
    java.lang.String[] data,
    java.lang.String formatDesc,
    gr.forth.ics.rdfsuite.services.DbSettings dbSettings)
```

**Description**

Imports in the underlying storage the specified namespaces and/or named graphs, along with their dependencies. The process is identical to the one described in 4.6.1 above, with two differences: first, this method also stores the namespace(s) and/or named graph(s) that the specified namespace(s) and/or named graph(s) depend on; second, if one of the requested namespace(s) or named graph(s), or the ones they depend on, is not found, the method searches for it across the network in the physical location that the URI of the namespace or named graph under question indicates (i.e., it uses eager resolution).

**Preconditions**

• The array of URIs has exactly the same number of elements as the array of data.
• Format is either TRIG or RDF/XML.
• If an RDF space declares its own URI (in a syntax-specific way), it must be the same as the respective entry in the array of URIs.

**Effects**

After the successful execution of the operation, the underlying storage will contain all supplied RDF spaces, identified by their respective URIs (see above), and all their dependencies, recursively. During storage, all redundant information in the original data is removed.

**Exceptions Thrown**

Identical to the exceptions described in subsection 4.6.1.

### 4.7 EXPORT SERVICE

#### 4.7.1 Fetch

**Signature**

```java
public java.lang.String[][] fetch(
    java.lang.String[] spaces,
    java.lang.String formatDesc,
    gr.forth.ics.rdfsuite.services.DbSettings dbSettings)
```

**Description**

This method is responsible for dumping into a byte sequence (in RDF/XML serialization or TRIG triple-based formats) the contents of the namespaces or named graphs given as input (spaces). The
user of the service needs only to specify the URIs (whose contents will be dumped) and the format to use (TRIG or RDF/XML).

**Preconditions**

Accepted formats are TRIG and RDF/XML.
The requested URIs should exist in the underlying storage.

**Effects**

Only the requested namespaces and/or named graphs are returned; not their dependencies. See subsection *Error! Reference source not found.* in this deliverable, or section 2.1 in [D5.2] for more details on the notion of dependencies among namespaces and named graphs.

**Exceptions Thrown**

The following exceptions are thrown by the service:

- *PostgresConnectionException:* this exception is thrown when the postgres driver is not found or when there is some other problem connecting to the postgres database.
- *ExporterDBException:* this exception is thrown when an exception query does not return the expected results because the database is not in the expected state.
- *ExporterOutputException:* this exception is thrown when an error appears in the writing to file/to string procedure of the exporter.

### 4.7.2 Fetch with Dependencies

**Signature**

```java
public java.lang.String[][] fetchWithDependencies(
    java.lang.String[] spaces,
    java.lang.String formatDesc,
    gr.forth.ics.rdfsuite.services.DbSettings dbSettings)
```

**Description**

Fetches the RDF contents of the requested URIs from the underlying storage and all their dependencies, using the requested format (TRIG or RDF/XML). The functionality is similar to the one described in 4.7.1, but, in this method, all transitively dependent namespaces or named graphs of the requested ones are also returned.

**Preconditions**

Accepted formats are TRIG and RDF/XML.
The requested URIs should exist in the underlying storage.

**Effects**

An error is raised if a requested URI, or any URI reachable recursively through dependencies, cannot be located (i.e., it has not been stored in the past). See subsection *Error! Reference source not found.* in this deliverable, or section 2.1 in [D5.2] for more details on the notion of dependencies among namespaces and named graphs. The returned array contains rows with RDF documents, in the requested format. Each row has exactly two string elements, the URI of a returned document, and the document itself.

**Exceptions Thrown**

Identical to the exceptions described in subsection 4.7.1.

### 4.7.3 Fetch with Data

**Signature**

```java
```

 IST-2006-033572 PUBLIC 36 / 100
public java.lang.String[][] fetchWithData(
  java.lang.String[] spaces,
  java.lang.String formatDesc,
  gr.forth.ics.rdfsuite.services.DbSettings dbSettings)

Description
Fetches the RDF contents of the requested URIs from the underlying storage, as fetch operation
does. Additionally, for every requested namespace, it returns a document containing every data
element that is related to that namespace (for example, a class instance of a class declared in that
namespace).

Preconditions
Accepted formats are TRIG and RDF_XML.
The requested URIs should exist in the underlying storage.

Effects
Only the requested namespaces and/or named graphs are returned; not their dependencies.

Exceptions Thrown
Identical to the exceptions described in subsection 4.7.1.

4.7.4 Fetch with Data and Dependencies

Signature
public java.lang.String[][] fetchWithDataAndDependencies(
  java.lang.String[] spaces,
  java.lang.String formatDesc,
  gr.forth.ics.rdfsuite.services.DbSettings dbSettings)

Description
Fetches the RDF contents of the requested URIs from the underlying storage, as
fetchWithDependencies operation does. Additionally, for every returned namespace, it returns
a document containing every data element that is related to that namespace (for example, a class
instance of a class declared in that namespace).

Preconditions
Accepted formats are TRIG and RDF_XML.
The requested URIs should exist in the underlying storage.

Effects
An error is raised if a requested URI, or any URI reachable recursively through dependencies cannot
be located (i.e., it has not been stored in the past). The returned array contains rows with RDF
documents (the requested ones plus the artificial documents, one for each namespace), in the requested
format. Each row has exactly two string elements, the URI of one returned document, and the
document itself.

Exceptions Thrown
Identical to the exceptions described in subsection 4.7.1.

4.8 QUERY SERVICE

Signature
- public java.lang.String query(
  java.lang.String queryString,
  java.lang.String preferredFormat,
  gr.forth.ics.rdfsuite.services.DbSettings dbSettings)
• public java.lang.String[] queryMultiple(
   java.lang.String[] queryStrings,
   java.lang.String preferredFormat,
   gr.forth.ics.rdfsuite.services.DbSettings dbSettings)

**Description**

The Query Service is responsible for executing RQL queries. It evaluates an RQL query, and returns the results in an RDF/XML or TRIG serialization as a bag of resources. The query results can contain both schema and data information from one or several namespaces and/or named graphs.

The Query method evaluates a single query and returns the results as described above. The QueryMultiple method evaluates multiple queries concurrently, for performance reasons.

The Query Service relies on the RQL Interpreter which is used for both parsing and executing the query at hand, as well as on its capability of spawning several instances of the RQL Interpreter in order to support concurrency control.

**Preconditions**

Accepted returned formats are TRIG and RDF.XML

**Effects**

In case of a syntactically wrong query, a result will be returned, in the specified format, containing an error message.

**Exceptions Thrown**

This service does not throw any exceptions.

### 4.9 UPDATE SERVICE

**Signature**

• public boolean update(
   java.lang.String rul,
   gr.forth.ics.rdfsuite.services.DbSettings dbSettings)

• public java.lang.Boolean[] updateMultiple(
   java.lang.String[] rules,
   gr.forth.ics.rdfsuite.services.DbSettings dbSettings)

**Description**

The Update Service is responsible for executing RUL updates involving one or several namespaces or named graphs, by evaluating a RUL statement and executing it. Updating includes construction, modification and deletion of objects in the repository and returns a value (true or false) for successful (commit) or unsuccessful (abort) execution.

The Update method evaluates a single RUL statement and returns the result as described above. The updateMultiple method evaluates multiple RUL statements, in a sequential and independent manner; for each update request, a value (true or false) is returned, indicating whether that particular request succeeded or not. Note that if the updates are mutually dependent, the results of this operation are unspecified, as no guarantee on the order of update execution can be made.

**Preconditions**

None.

**Effects**

In case of a syntactically wrong RUL statement, a result will be returned, in the specified format, containing an error message.

**Exceptions Thrown**

This service does not throw any exceptions.
4.10 EXAMPLE OF KNOWLEDGE EVOLUTION

For the case where knowledge is expressed formally in the form of ontologies and metadata, it would be advantageous to support declarative languages for bulk metadata updates. This is a benefit of adopting an advanced repository (this task could be very laborious and uncontrolled if RDF were stored only as plain files). An example is shown in the next figure. Due to the division of Yugoslavia several changes have to be made.

![Diagram showing metadata evolution](image)

Figure 4-1 Example of Metadata Evolution

To update the data layer of the knowledge base (as shown at the bottom part of Figure 4-1) we could use the following program (sequence) of RUL statements:

- INSERT Country(&Slovenia), Currency(&SIT), currency(&Slovenia,&SIT)
- REPLACE inCountry(&Maribor, &Yugoslavia->&Slovenia)
- REPLACE value(X, Y->Y*0.68), withCurrency(X, Z->&SIT)
  FROM {X}value{Y}, {X}withCurrency{Z}
  WHERE Z =& YUM
- DELETE Country(&Yugoslavia), Resource(&Yugoslavia), Currency(&YUM), Resource(&YUM)
This would update all prices that are expressed in the (currently obsolete) YUM currency. This example stresses the advantages of having an advanced semantic web repository and supporting updating knowledge in a declarative way.

The above program could be passed as input to the Update Service that was specified earlier.

Other approaches to support evolution are currently being investigated [Tzitzikas, AIA’2007].

Comparison functions are also very important (e.g. when different versions of ontologies come up). This is also a direction that is worth research and development work.

### 4.11 COMPARISON SERVICES

The Comparison Service is responsible for comparing two collections of name or graph spaces (KBs) already stored in the repository and compute their delta in an appropriate form. The result of the comparison is a "delta" (or "diff") describing the differences between the two collections of name or graph spaces, i.e., the change(s) that should be applied upon the first in order to get to the second (see the next figure for an example). The intended use of the service is the comparison of two different versions of the same name or graph space to identify their differences; comparing unrelated name or graph spaces (i.e., name or graph spaces which are not different versions of the same name or graph space) would give results which have no intuitive meaning.

Notice that the problem of comparing two name or graph spaces is very different from the problem of comparing the source files (e.g., TRIG files) which describe them. This is true because

- A name (or graph) space carries semantics, as well as implicit knowledge which is not part of the source file.
- There are alternative ways to describe syntactically the same construct (triple) in a name or graph space, which could result to erroneous differences if resorting to a source file comparison method.
- And (c) source files may contain irrelevant information, e.g., comments, which should be ignored during the comparison.

It is clear by the above analysis that the comparison should be based on semantic, rather than syntactic considerations, so the comparison service will be based on the comparison of the
triples contained in the name or graph spaces. There are alternative methods for computing a
semantic delta between name or graph spaces. In particular, the implicit knowledge (i.e., the
inferred triples) contained in the two name or graph spaces may or may not be taken into
account, leading to the following four cases:

- **Delta Explicit (Δe)**: Takes into account only explicit triples
  \[ Δ_e(K \rightarrow K') = (Add(t) \mid t \in K' - K) \cup (Del(t) \mid t \in K - K') \]

- **Delta Closure (Δc)**: Takes also into account inferred triples
  \[ Δ_c(K \rightarrow K') = (Add(t) \mid t \in C(K') - C(K)) \cup (Del(t) \mid t \in C(K) - C(K')) \]

- **Delta Dense (Δd)**: Returns the explicit triples of one KB that do not exist at the closure of the other KB
  \[ Δ_d(K \rightarrow K') = (Add(t) \mid t \in K' - C(K)) \cup (Del(t) \mid t \in C(K) - C(K')) \]

- **Delta Dense & Closure (Δdc)**: resembles Δd regarding additions and Δc regarding deletions
  \[ Δ_{dc}(K \rightarrow K') = (Add(t) \mid t \in K' - C(K)) \cup (Del(t) \mid t \in C(K) - C(K')) \]

In the above bullets the operator \( C(\cdot) \) stands for the consequence operator, which is a function
producing all the consequences (implications) of a name or graph space \( K \), i.e., all the inferred
triples of \( K \). In the example in Figure 5.1, only the explicit knowledge is taken into account
in the comparison, so the shown result corresponds to Δe. If the implicit knowledge was also
taken into account, the result would be different (e.g., Δc, Δd and Δdc, would not report the
addition of the [C IsA A] triple).

One of the main properties that we intuitively expect to hold in a comparison function is that its
output, when applied upon the first name or graph space, should give the second; this
property is called correctness. In order to study which of the four delta functions guarantees
correctness, we should first determine what it means for the output of the diff service to be
"applied" upon the first name or graph space. The latter issue is related to the semantics of the
update operations considered, i.e., a formal description of how the output of the diff should be
"applied" upon the name or graph space.

There are three options in this respect, namely: (a) that the operations (additions and deletions
of triples) that are included in the delta are viewed as plain set additions and deletions (plain
semantics - Up); (b) that they are coupled with redundancy elimination and computation of
logical implications (inference and reduction semantics - Uir); or (c) that they are handled
using the change semantics introduced by the Change Service (change service semantics
-Ucs).

Using this definition of update semantics, in paper [S5] it was shown that only certain pairs of
delta functions with update semantics are correct, namely: (Δe,Up), (Δdc,Uir) and (Δc,Uir).
Most existing comparison tools rely on the (Δe,Up) pair.

Another critical consideration is related to the size of the delta; in this respect, delta dense
(Δd) is best, compared to any other delta function, whereas Δdc gives smaller in size delta
than Δc; on the other hand, Δe and Δc are incomparable. Notice however that, as we saw
above, Δd (the smallest possible delta) does not guarantee correctness.
In our implementation, we don't adopt any particular policy regarding the "correct" or "best" delta function; in particular, the delta function to be used is just a parameter of the service, and the caller is assumed to understand the implications of using any particular delta function.

**Signature**

```java
public gr.forth.ics.rdfsuite.services.Delta diff(
    java.util.Collection<java.lang.String> namespacesOrGraphspaces1,
    java.util.Collection<java.lang.String> namespacesOrGraphspaces2,
    gr.forth.ics.rdfsuite.services.DeltaFunction deltaFunction,
    gr.forth.ics.rdfsuite.services.DbSettings dbSettings)
```

**Description**

The Comparison Service is responsible for comparing two collections of namespaces or named graphs (KBs) already stored in the repository and compute their delta in an appropriate form. The result of the comparison is a “delta” (or “diff”) describing the differences between the two collections of namespaces or named graphs, i.e., the change(s) that should be applied upon the first in order to get to the second. It should be noted that these collections are expanded with those namespaces or named graphs that they respectively depend on.

Notice that the redundancy elimination feature of the Import Service could cause unexpected results in the behaviour of the Comparison Service, for those diff functions that use explicit information, which is affected by redundancy elimination (see [S5]). This is true only if redundancy elimination is applied, e.g., if one imports the models in the database (from the input files) and compares the results; it will not happen if the models are compared in the main memory, before being loaded into the database.

It should be emphasized that the comparison is not performed upon the namespaces and named graphs in the input only (namespacesOrGraphspaces1 and namespacesOrGraphspaces2), but also upon the namespaces and named graphs that they depend on. In other words, the compared conceptualizations occur by taking the union of the triples in the URIs indicated by `nameGraphSpaceURI1` (and `nameGraphSpaceURI2`) plus the triples in the namespaces or named graphs that the input namespaces or named graphs depend on.
Figure 11: Sequence Diagram for the Comparison Service

Figure 11 depicts the process taking place in the Comparison Service in a sequence diagram.

**Preconditions**

- Namespaces and named graphs that are described in `nameGraphSpaceURI1` and `nameGraphSpaceURI2` should already be stored in the repository.
- Possible values for the `deltaFunction` parameter are: “delta_explicit”, indicating that Delta Explicit (Δε) should be used; “delta_closure”, indicating that Delta Closure (Δc) should be used; “delta_dense”, indicating that Delta Dense (Δd) should be used; and “delta_dense_and_closure”, indicating that Delta Dense and Closure (Δdc) should be used. For details on those delta functions refer to [S5].

**Effects**

None.

**Exceptions Thrown**

This service does not throw any exceptions.

### 4.12 SWKM Main Memory Model

SWKM Model is a Main Memory RDF/S Management System. Unlike most known RDF parsers, it provides both an Object-based Model and a Triple-based Model for Main Memory RDF manipulation. The Model is not intended to be either a persistence storage model or a model that can be updated programmatically in main memory. For this purpose one should use the SWKM Services.
An important feature of the SWKM Model is its capability of handling both namespaces and graphspaces. For every namespace of the model, the triples in it are saved and as a result, they can easily be retrieved. A namespace is used programmatically as a container where all the relevant information to an actual RDF namespace is kept. Namespaces hold information (references) to the namespaces they depend on. A graphspace is defined as an arbitrary but defined collection of triples. Again, graphspaces hold information (references) to the namespaces or graphspaces they depend on. This provides the ability to save information either at the model, the namespace or the graphspace level, depending on its nature.

In SWKM Model a node is either a literal or a resource. A Hash Map that holds all the literals existing in the model is used. If the node is a resource it can be one of class, property, class instance or property instance. Depending on its type the node is saved in separate Hash Maps at the model level (class instances and property instances) or at the namespace level (classes and properties), with each Hash Map holding one type of resources and using as key its (unique) URI and as value the corresponding object.

After the nodes are saved, the triple is created as an object holding references to the nodes involved and to the namespace and/or graphspace that it might belong (if the same triple is found belonging to another graphspace then only a reference is added) and is saved either in the model or in the namespace it belongs. Triples are again typed against the RDF/S basic properties:

- Subclass
- Subproperty
- Domain
- Range
- Comment
- Label
- Seals
- Isdefinedby
- Member
- Type

If the predicate belongs in none of these categories, it is considered to be a property instance. All of these are typed as subclasses of the class Triple and their typing information can be exploited programmatically. For each of the mentioned types, there are six hash maps that contain the corresponding triples as values, having as keys the subject, the object or a multi key of all their possible combinations respectively. This has as a result larger memory usage but faster retrieval times for the triple view. Moreover separating the triples and providing object representations for each predicate type facilitates work on diff and evolution (and other algorithms that use the triple based representation) of RDF models, since it allows for easier identification and manipulation of the triples.
On the other hand, SWKM Model provides a complete object view on the RDF model at hand. This means that nodes can be retrieved fast (since they are saved in Hash Maps) and moreover all information available in the model can be retrieved by the API methods provided at the object level.

4.12.1 About API Access Methods.

The SWKM Model API is a rich API. The developer has the possibility to choose between using the triple based API and the object based API to retrieve the required information. In both cases a wealth of access methods is offered.

In the triple based view a retrieval method for all triples of the model, as well as one for all triples with some subject, predicate and/or object are offered. To retrieve all triples of a model, all values of the hash maps used are returned. For this operation, all the maps hashed on the subjects are used. When searching for triples by subject, predicate and/or object, the respective hash maps are used. For example, when the predicate is given, the two hash maps of the corresponding property or property instance are used: the one hashed on the subjects if only the subject and the predicate are given and the one hashed on object if the predicate and the object are given. Finally, if the predicate is not given, the desired values of the hash maps of all the predicates are added (e.g. if only the subject is given as a parameter, the hash maps, hashed on the subjects will be used) and an iterator on them will be returned.

A rich set of variations of these methods are also provided. For instance, a user can choose not to include in the results of the "find all triples" method the triples contained in RDF/S. On another example one can retrieve triples searching for them based on their (string) URI and not the object representation of the node. In between the triple and the object based view methods one can retrieve all triples associated with a specific namespace or graphspace by giving its unique URI. This is extremely useful for applications manipulating namespaces or graphspaces as one object.

In the object based view one can retrieve all the necessary information starting at any of the objects without having to retrieve and process any triples in between. This provides, for example, easy access to subsumption relationship information like subClassOf, superClassOf, subPropertyOf and superPropertyOf (both direct and all the descendants/ancestors). This methods work recursively by finding the direct descendants (or ancestors) and then their direct descendants or ancestors. The same way access information on in which triple this class or class instance serves as a subject (domain) or an object (range) can be provided.

The object based view offers unique capabilities to the programmer using the API that allow objectification of the RDF information in a consistent way providing objects that represent a full RDF Model. Since this can be combined when needed with information retrieved through the triple based access methods, the programmer has a wealth of methods and ways to manipulate the same RDF information. This combined capability of representing and manipulating RDF information in main memory both as triples and as full blown objects is to the best of our knowledge unique in SWKM.

4.12.2 More Detailed Description
As we can see in Figure 12, SWKM employs two additional abstractions as containers of the RDF/S model information: namespaces and named graphs. In this context, an RDF/S model consists of namespaces and named graphs collections. A namespace gathers all the class and property nodes along with the respective triples that are defined in an RDFS schema associated to the model while a named graph keeps track of all edges relating nodes through various kinds of properties. In SWKM, the `subject`, `predicate` or `object` of a triple are Java objects whose state and type information is determined by the triples of an RDF/S model.

In turn a triple is implemented as a Java object holding references to the participating nodes. At the namespace level four maps named `keyToTypetriples`, `keyToSubclasstriples`, `keyToDomaintriples`, `keyToRangetriples`. 

![Figure 12: SWKM Data Structures](image)

<table>
<thead>
<tr>
<th>Triple View</th>
<th>Object View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model / Namespace / GraphSpace</td>
<td></td>
</tr>
<tr>
<td>Model / Namespace / GraphSpace {</td>
<td></td>
</tr>
<tr>
<td>HashMap&lt;URI, Namespace&gt; namespaces;</td>
<td></td>
</tr>
<tr>
<td>HashMap&lt;URI, Graphspace&gt; graphspaces;</td>
<td></td>
</tr>
<tr>
<td>HashMap&lt;Key, HashSet&lt;PropertyInstance&gt;&gt; keyToPropertyInstancetriples;</td>
<td></td>
</tr>
<tr>
<td>//where Key ∈ {s, p, o, &lt;s,p&gt;, &lt;s,o&gt;, &lt;p,o&gt;, &lt;s,p,o&gt;}</td>
<td></td>
</tr>
</tbody>
</table>

| Typetriple: triple with p = “rdf:type” |
| Subclasstriples: triple with p = “rdfs:subClassOf” |
| Domaintriples: triple with p = “rdfs:domain” |
| Rangetriples: triple with p = “rdfs:range” |

| s: subject |
| p: predicate |
| o: object |
keyToRangeTriples are used to store schema related triples discriminated by their predicate, i.e., using a different map for each possible property of the RDF/S. The Maps used are actually MultiHashMaps, which store the value of the HashMap’s <key:value> pair in another HashMap. Moreover these Maps are indexed by subject, object and by subject and object (meaning that actually three HashMaps are kept per RDF/S property). Class and property instance related information are kept at the model level in the classInstances and propertyInstances HashMaps respectively. Actually, for the propertyInstances, six maps are kept since subject or predicate or object and all their possible combinations are used as multikeys for faster search. Triples are treated as objects, subclasses of the root class Triple, allowing for programmatic access to them through common interfaces and methods. Class Triples (Figure 13) instantiations carry also information about the namespace and/or the named graph they belong to. Finally, separating the triples and providing object representations for each RDF/S property type make easier the development of sophisticated algorithms for RDFS schema evolution and comparison.

Nodes are also stored as objects either at the namespace (when they are schema related) or at the model (when they are instance related) level. Two HashMaps, classes and properties, are used to store the objects representing the classes and the properties of each namespace and kept at the namespace level. On the other hand, one additional HashMap is used to store resources (classInstances) at the model level. Nodes carry within available information that is computed at run time, e.g., a node of type Class carries information about its subclasses and its superclasses (in both cases immediate and all), a node of type Class Instance carries information about its classification, etc; this information is computed from the corresponding triples once. Finally, namespaces and named graphs themselves hold dependency information, i.e., which namespace or which namespace and named graph respectively they might depend on.

Using the SWKM API, the developer has the possibility to choose between the triple-based and the object-based views to access the same information represented in an RDF/S model. In the triple-based view, a retrieval method for all triples of the model, as well as one for all triples with some subject, predicate and/or object are offered. To retrieve all triples of a model, all values of the previously described maps are returned. For this operation, all the maps hashed on the subjects are used. When searching for triples by subject, predicate and/or object, the respective HashMaps are used. A rich set of variations of these methods is also provided. For instance, a user can choose not to include in the results of the “find all triples” method the triples contained in the default RDF/S namespace or one can retrieve triples searching for them either by their (string) URI or by the object representation of the
node. As already stated, we can also retrieve triples that belong to a specific namespace or named graph.

In the object-based view, the objects carry information firstly through their types. Additional information is computed, like `subClassOf`, `superClassOf` and `subPropertyOf`, `superPropertyOf` information for classes and properties respectively, which then can be retrieved directly from the object’s members in constant time by returning an Iterator on the structure(s) that hold it. The same way information on which triples the object serves as domain, range or predicate is also kept. The object-based view offers an objectification of the RDF information in a consistent way. Additionally, in any step this can be combined with information retrieved through the triple-based access methods. Actually, one can seamlessly move between the two views since the objects mainly serve as triple nodes and can be reached through the triples and the triples can be found based on the HashMap keys which are build based on the objects. This combined capability of representing and manipulating RDF information in main memory both as triples and as full blown objects is to the best of our knowledge unique in SWKM Main Memory RDF Model.

## 5 SOFTWARE COMPONENT

### 5.1 ITERATIONS

The main functionality of each iteration and some testing/validation methods are described in the following table. The table includes not only the Registry-related KM services but also the basic KM services on which they depend.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Functionality</th>
<th>Motivation</th>
<th>Testing/Validation Method(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>First version of the Basic SWKM Services. Options: 1/ Full installation 2/ Installation of only the client.</td>
<td>The high level knowledge management services (e.g. RepInfoGap Manager) will be based on these. In addition, the rest CASPAR components may exploit these basic services.</td>
<td>I1TC1/ Ability to build a SW repository offering persistence, validation and query and update services. I1TC2/ Ingestion test: one partner from the testbeds (e.g. IRCAM) uses these services in order to feed the repository with some data (e.g. with the CIDOC CRM ontology and descriptions of some indicative objects with respect to that ontology). I1TC3/ The component Access Manager uses these services to offer some content-based access services. E.g. provides some provenance queries assuming the CIDOC CRM ontology.</td>
</tr>
<tr>
<td>I2</td>
<td>(2a) Revised version of the Basic SWKM Services including a design and a first implementation of a Main Memory Model. (2b) First version</td>
<td>RepInfo Gap Manager is instrumental for providing intelligibility-aware services</td>
<td>I2aTC1/ Ability to support some forms of knowledge evolution (recall the example of Yugoslavia). I2aTC2/ Provision of an API for managing Semantic Web data in main memory. I2bTC2/ Ability to implement the examples described in</td>
</tr>
<tr>
<td>I3</td>
<td>Second version of Gap Manager</td>
<td>Interaction with other caspar components. GapManager should actually receive requests from other components. It is not expected to call other components.</td>
<td>I3TC1/ The registry or POM feeds the KM Repository (with modules and dependencies). I3TC2/ The component Preservation Orchestration Manager uses the RepInfoGapManager services.</td>
</tr>
<tr>
<td>I4</td>
<td>Diff over Knowledge Bases</td>
<td>Support of knowledge evolution is important and it should be demonstrated.</td>
<td>I4TC1/ It takes as input two versions of one ontology (say V1 and V2 of CIDOC CRM). The tool should be able to identify the changes and derive a set of change operations which could be applied on a knowledge repository on CIDOC CRM V1 in order to reach CIDOC CRM V2.</td>
</tr>
</tbody>
</table>

Table 1 Functionality of each iteration

The development follows the schedule and all scheduled functionality has been developed and tested. Currently we are at the end of I3. In addition auxiliary tools for aiding data holders have been developed.

5.2 MORE DETAILS ON TESTING

Each test scenario is checked by writing the corresponding JUnit tests. In this way it is possible to run them periodically and automatically. This is important for ensuring the correctness of the code as the project proceeds and new functionality is added. For each of the testing scenarios of the previous table a more detailed set of test cases are given in the following table. Of course, more test cases will be designed and developed as the project proceeds.

<table>
<thead>
<tr>
<th>HIT C1</th>
<th>N o</th>
<th>Ability to build a SW repository offering persistence, validation and query and update services.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Try to import a valid (syntactically and semantically) expression of CIDOC CRM in RDFS to the repository. The import request should succeed and the correct internal structures of the repository are created.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Try to import an invalid (syntactically or semantically) expression of CIDOC CRM in RDFS to the repository. The import request should fail and the KM repository should remain intact. This test will ensure that the import requests have transaction semantics.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Try to import a namespace B that extends a namespace A that is already imported. The request should succeed. If A is not already imported the request should fail.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Try to export from the repository an ontology that has already been imported. A valid output file should be returned. The output file should be possible to be imported to the KM repository.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Try to export from the repository an ontology with all its dependencies (i.e. with all other ontologies that it extends). The result should be a set of files that someone should be able to import to a new KM repository.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Try all combinations of pre/post conditions of the specification of the basic services (that are described)</td>
</tr>
<tr>
<td>Task</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td></td>
</tr>
</tbody>
</table>
| 1ITC2 | Ingestion test: one partner from the testbeds (e.g. IRCAM) uses these services in order to feed the repository with some data (e.g. with the CIDOC CRM ontology and descriptions of some indicative objects with respect to that ontology).  
Try to import some valid instantiations of CIDOC CRM expressed in RDF. The import request succeeds and the correct internal structures of the repository are created. A failure should be reported if the file to be imported is syntactically or semantically incorrect.  
Try to import some valid instantiations of CIDOC CRM expressed in RDF. The import request should succeed and the correct internal structures of the repository should have been created.  
Use the query language that verify that a class c1 is subclass of a class c2 for the case where c1 and c2 have been defined in different namespaces and c2 has been defined as an extension of c1 in the corresponding RDF file that was imported. |
| 1ITC3 | The component Access Manager uses these services to offer some content-based access services. E.g. provides some provenance queries assuming the CIDOC CRM ontology.  
Some indicative queries assuming CIDOC CRM are formulated. The KM repository may have stored instances according to a specialization of CIDOC CRM (e.g. wrt a schema B that extends the concepts of CIDOC CRM). The result of the query should not be empty. |
| 12aTC1 | Ability to support some forms of knowledge evolution (recall the example of Yugoslavia).  
Test the scenario of Yugoslavia. Formulate a number of queries that ensure that the KB has been updated. Ensure transaction semantics. |
| 12aTC2 | Testing the API for managing Semantic Web data in main memory.  
Try exporting a namespace that is already stored in the KM repository. From the returned file it should be possible to create a main memory model (MMM) that an application programmer could use in order to build an application. Specifically the MMM should allow someone to get the classes, the properties, the resources and the property instances that are defined in the exported file. |
| 12bTC2 | Ability to implement the examples described in this deliverable regarding intelligibility gaps. Definition of modules, profiles, dependencies, etc. Demonstrating examples that users with different profiles get different responses.  
For each of the methods described in Section 0 a test case will be created on the basis of the pre/post-conditions that are already specified. |
| 13TC1 | The registry or POM feeds the KM Repository (with modules and dependencies).  
The repository stores and responds to the requests according to the specification (related: 12bTC2). The messages received from POM should be manageable by RepInfoGapManager.  
POM is notified by the ReplInfoRegistry about a change of a module. POM should be able to call the appropriate methods of CKM in order to identify the affected DC profiles. This will ensure that the design specification is complete. |
| 14TC1 | It takes as input two versions of one ontology (say V1 and V2 of CIDOC CRM). The tool should be able to identify the changes and derive a set of change operations which could be applied on a knowledge repository on CIDOC CRM V1 in order to reach CIDOC CRM V2.  
This scenario involves a lot of research. For this reason this scenario has a “WANT” priority (according to Must Should Could Want scheme for setting priorities)  
1. Take as input two namespaces A and B expressed in RDFS  
2. Compute the Diff between two A and B as a sequence of change operation DU  
3. We have to test whether the application of DU on a knowledge repository that contains
A will result to one namespace that is semantically equivalent to A
Several experiments (with various different pairs of A and B namespaces) will be conducted.

Table 2 Test Cases

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Input</th>
<th>Start Date</th>
<th>End Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>All CASPAR deliverables</td>
<td>M6</td>
<td>M23</td>
</tr>
<tr>
<td>I2</td>
<td>D2101B</td>
<td>M18</td>
<td>M27</td>
</tr>
<tr>
<td>I3</td>
<td>Revised architecture, Comments from the 2nd year review, Data holders needs</td>
<td>M28</td>
<td>M31</td>
</tr>
<tr>
<td>I4</td>
<td>Instantiations of CIDOC CRM and its extensions.</td>
<td>M28</td>
<td>M35</td>
</tr>
</tbody>
</table>

Table 3 Schedule of Iterations
5.3 CASPAR KM INSTALLATION

5.3.1 Required software components

The Prototype of the Knowledge Manager is based on Java based web services (SWKM services from now on) running on top of a working Glassfish installation. This implies that a working Java Development Kit installation (at least Java 5.0) should be already present or downloaded and installed from [http://java.com/java/download/index.jsp](http://java.com/java/download/index.jsp).

![Diagram of SWKM services](image)

**Figure 1. The various software components required by the SWKM services**

In detail the required components for SWKM services to run and their various connections and dependencies as well as the available services after deployment are depicted in Figure 1.
The latest releases of software and documentation for the Semantic Web Knowledge Middleware (basic services, client and main memory model) is continuously available at http://athena.ics.forth.gr:9090/SWMK/.

The Gap Manager is based on the Client described previously. The latest version of the software can be downloaded from http://wiki.casparpreserves.eu/bin/view/Main/2103GapManager

It is structured according to the CASPAR best practices document and it is accompanied by comments, examples and unit tests.

5.4 A POSSIBLE DEPLOYMENT

Figure 5-1 illustrates a possible deployment for all KM-related components. A KM client will be provided that is able to connect with the basic SWKM services. This will allow other components to use easily these services.

For the same reason the high level KM services will be provided as a client application.

![Deployment Diagram](Image)

Figure 5-1 Proposed deployment for KM-related components

5.5 SUPPORTED CHARACTERS IN SEMANTIC WEB MIDDLEWARE MANAGER (SWKM)

Semantic Web Knowledge Middleware (SWKM) supports Unicode RQL/RUL query strings. Not every character is allowable though:

A URI can have the following characters only:

# = . / : ? _ ~
0..9
a..Z
A..Z

An RDF literal can be composed of the following characters:
0x0009 (tab character)
{0x0010..0xFFFF} (space character and beyond)

Care should be taken when the query in the client side is not stored in a Unicode-capable format (a java.lang.String with the appropriate characters should be fine). Furthermore, the query strings needs to be sent over the network, so it is up to the implementation of the client stub to preserve the queries’ characters. JAX-WS has been tested for implementing the client side with no character problems whatsoever.

SWKM operations that serve XML will take care to appropriately encode characters that cannot be contained in a valid XML document. These characters are (followed by their respective encodings):
< → &lt;
> → &gt;
" → &quot;
' → &apos;
& → &amp;
6 USING THE SWKM SERVICES: EXAMPLES

The SWKM client is a collection of Java classes and interfaces that can be used to contact the SWKM web services in a concise and natural way.

Each web service is modeled as a Java interface. These interfaces reside in swkm-services-api.jar, in the package gr.forth.ics.rdfsuite.services. The mapping of interface names and services is given below:

<table>
<thead>
<tr>
<th>WEB SERVICES</th>
<th>WSDL URL PATHS</th>
<th>INTERFACES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query Service</td>
<td>/query?wsdl</td>
<td>QueryHandler</td>
</tr>
<tr>
<td>Update Service</td>
<td>/update?wsdl</td>
<td>UpdateHandler</td>
</tr>
<tr>
<td>Import Service</td>
<td>/importer?wsdl</td>
<td>Importer</td>
</tr>
<tr>
<td>Export Service</td>
<td>/exporter?wsdl</td>
<td>Exporter</td>
</tr>
<tr>
<td>Comparison Service</td>
<td>/diffGenerator?wsdl</td>
<td>DiffGenerator</td>
</tr>
</tbody>
</table>

The javadoc specification of the interfaces can be used as a reference for interacting with the respective web services. We will here provide examples of how to get instances of the interfaces, as well as several small examples of using them.

6.1 BOOTSTRAPPING

The first step to access the services is to create a client to them. The client can be found in the SwkmClient.jar, by importing the package gr.forth.ics.swkmclient. The goal is to create a gr.forth.ics.swkmclient.Client instance, which provides references to interface instances that can interact with the web services.

There are various ways to get a Client instance, all residing as static factories in class ClientFactory. For a detailed reference, see the specification of that class. The two most common ways are either through a properties file (“services.properties”) located on the classpath, or directly by providing the host’s URL, its port, and the web application context that provides the web services. To avoid proliferating configuration files in an application, the latter method is preferred, which allows an application having centralized configuration.

Another piece of configuration is which database settings to use. The SWKM services have the feature that a single deployment can talk to as many databases as required, without needing a separate deployment for each configuration. As the web services are strictly stateless for scalability reasons (i.e., they don’t maintain state between subsequent requests of the same client), this is implemented by including the relevant database configuration to be used by the service, inside the request. Each deployment of the web application defines its own default database configuration, and each setting can be overridden independently by a client – all not defined configuration settings are inherited from the default configuration. For example, one could simply define the database name to be used, and nothing else, so the user and password configured by default would be used to access this database.
Putting it all together:

```java
import gr.forth.ics.swkmclient.*;

This creates a client by loading a “services.properties” file from the classpath, which will use the
default database settings of the web application.

ClientFactory.newClientByConfigurat
.on27,  
SwkmMiddlewareWS").withDefaultSettings();

Or to create a client by providing explicitly the endpoint location:

Client client = ClientFactory.newClientByConfiguration(  
  "http://139.91.183.8", 3027,
  "SwkmMiddlewareWS").withDefaultSettings();

The Client interface is an access point to the interfaces specified in package
gr.forth.ics.rdfsuite.services”.
In order to create a client that creates requests for a specific database, this would be used:

```java
import gr.forth.ics.swkmclient.*;
import gr.forth.ics.rdfsuite.services.*;

DbSettings customDbSettings = new DbSettings();
customDbSettings.setDbName("marketak_db");
customDbSettings.setUsername("swkm");

Client client = ClientFactory.newClientByConfiguration(  
  "http://139.91.183.8", 3027,
  "SwkmMiddlewareWS").with(customDbSettings);
```

6.2 CLIENT’S METHODS

As mentioned earlier, a Client instance is an access point to several interfaces in
gr.forth.ics.rdfsuite.services”. These group the operations of each web service. They are:

- **Importer** accessed by `client.importer()`
- **Exporter** accessed by `client.exporter()`
- **QueryHandler** accessed by `client.query()`
- **UpdateHandler** accessed by `client.update()`
- **DiffGenerator** accessed by `client.diffGenerator()`
6.3 KNOWLEDGE REPOSITORY USAGE EXAMPLE

All examples assume the following import statements:

```java
import gr.forth.ics.swkmclient.*;
import gr.forth.ics.rdfsuite.services.*;
import gr.forth.ics.rdfsuite.services.utils.*;
import java.util.*;
```

6.3.1 Importer Example 1

This example loads in memory the CIDOC.rdfs document, along with a URI to describe it.

```java
RdfDocument cidoc = new RdfDocument(
    "http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf#",
    IOUtils.readString(new File("data/cidoc/CIDOC.rdfs"),
        Format.RDF_XML);
```

Creating the client and using the default configuration...

```java
Client c =
    ClientFactory.newClientByConfiguration().withDefaultSettings();
```

Start with a clean database, for demonstration purposes. Of course, this functionality can be disabled through configuration (by setting the EraseDatabase property to false).

```java
c.debug().eraseDatabase();
```

Getting a handle to the Importer interface.

```java
Importer importer = c.importer();
```

Lets store CIDOC. It does not have any dependency that must be looked up on the network, so we specify Deps/WITHOUT (“without dependencies”).

```java
importer.store(Arrays.asList(http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf#), Deps.WITHOUT);
```

6.3.2 Importer Example 2

Create a client by reading services.properties from classpath and using the default database configuration of the server.

```java
Client c =
    ClientFactory.newClientByConfiguration().withDefaultSettings();
```

Load in memory the FRBR schema file, that extends CIDOC.

```java
RdfDocument frbr = new RdfDocument(
```
Let's store CIDOC first, to have something to export.

```java
    IOUtils.readFileAsString(new File("data/cidoc/frbr.rdf\$"),
    Format.RDF_XML);
```

Start with a clean database, for demonstration purposes. As mentioned, this functionality can be disabled through configuration.

```java
    c.debug().eraseDatabase();
```

Getting a handle to the Importer interface.

```java
    Importer importer = c.importer();
```

Let's store FRBR. FRBR needs CIDOC, and we didn't store it yet, nor we provide it to the importer service. Instead, we will request any missing dependencies to be looked up on the network (but using their base URIs as URLs), by specifying Deps.WITH. CIDOC will also be stored after this operation.

```java
    importer.store(Arrays.asList(frbr), Deps.WITH);
```

### 6.3.3 Exporter Example 1

Let's store CIDOC first, to have something to export.

```java
    ImporterExample1.main(args);
```

Create a client by reading services.properties from classpath and using the default database configuration of the server.

```java
    Client c =
    ClientFactory.newClientByConfiguration().withDefaultSettings();
```

Get a handle to the Exporter interface.

```java
    Exporter exporter = c.exporter();
```

The base URI of CIDOC.

```java
    String cidocUri = "http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf#";
```

Get CIDOC document as TriG. We don't want any dependencies to be fetched, or data instances of the specified schema belonging to other named graphs, so we specify Format.TRIG, Deps.WITHOUT and Data.WITHOUT respectively.

```java
    Map<String, RdfDocument> docs = exporter.fetch(
```
Arrays.asList(cidocUri),
    Format.TRIG, Deps.WITHOUT, Data.WITHOUT);

RdfDocument cidoc = docs.get(cidocUri);

Print out exported CIDOC.

System.out.println(cidoc.getContent());

6.3.4 Exporter Example 2

Let's store CIDOC and AVIS first, to have something to export.

ImporterExample2.main(args);

Create a client by reading services.properties from classpath and using the default database configuration of the server.

Client c =
    ClientFactory.newClientByConfiguration().withDefaultSettings();

Get a handle to the Exporter interface.

Exporter exporter = c.exporter();

The base URI of CIDOC:
String cidocUri = "http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdfs#";

The base URI of FRBR:
String frbrUri = "http://cidoc.ics.forth.gr/rdfs/FRBR.rdfs#";

We will fetch FRBR, and specify that we also want its dependencies (Deps.WITH). This will cause CIDOC to be fetched as well.

Map<String, RdfDocument> docs = exporter.fetch(
    Arrays.asList(frbrUri),
    Format.TRIG, Deps.WITH, Data.WITHOUT);

RdfDocument frbr = docs.get(frbrUri);
RdfDocument cidoc = docs.get(cidocUri);

Print out exported CIDOC:

System.out.println(cidoc.getContent());
6.3.5 Query Example

Lets store CIDOC first, to have something to query.

`ImporterExample1.main(args);`

Create a client by reading services.properties from classpath and using the default database configuration of the server.

`Client c = ClientFactory.newClientByConfiguration().withDefaultSettings();`

Get a handle to the QueryHandler interface.

`QueryHandler queryHandler = c.query();`

Find all available classes by a trivial RQL query.

`String result = queryHandler.query(Format.RDF_XML, "classes");`

Get a list with the classes and print it.

`List<String> classes = QueryUtils.list(result, Format.RDF_XML); System.out.println(classes);`

6.3.6 Update Example

Lets store CIDOC and some class instances first, to have something to update.

`ImporterExample1.main(args);`

Create a client by reading services.properties from classpath and using the default database configuration of the server.

`Client c = ClientFactory.newClientByConfiguration().withDefaultSettings();`

Get a handle to the UpdateHandler interface.

`UpdateHandler updateHandler = c.update();`

Insert, through a URL query, an event.

`boolean succeeded = updateHandler.update("Insert Task(&somethingHappened#)");`
Let’s verify that it was stored.

String queryResult = c.query().query(Format.RDF_XML, "Task");
List<String> instances = QueryUtils.list(queryResult, Format.RDF_XML);

Now this list will contain an element “somethingHappened#”.

System.out.println(instances);

7 USING THE SWKM SERVICES: FINDING AIDS

This Section presents how the CASPAR Finding Aids (FA) use the above described SWKM services. The structure is as follows:

- Section 7.1 highlights the OAIS context in which the FA is defined and used by other OAIS services.
- Section 7.2 gives the conceptual view of FA, presenting the main concepts underlying the FA architecture; these two Sections have already been included in a previous deliverable [D2301] and are here recalled for the sake of self-containedness.
- Section 7.3 gives the logical view of the FA, and is an update of the analogous view presented in [D2301].
- Section 7.4 gives the details on how the Finding Manager uses SWKM.

A very simple example of Descriptive Information schema and Descriptive Information object managed by FA is given in Section 9.2. More complex examples can be found on the project Wiki at http://wiki.casparpreserves.eu/bin/view/Main/TaskId2305.

7.1 INTRODUCTION TO FINDING AIDS

Finding Aids CASPAR component corresponds to the Access OAIS entity, that make the archival information holdings and related services visible to Consumers. As the Figure below shows (Figure 2), it is related with the User Application to aid user to locate relevant data.

FA is also associated with the Persistent storage system to collect Descriptive Information (DescInfo) about AIPs.
In concrete, Finding Aids is a component that allows users to search for CASPAR AIPs based on the associated Descriptive Information objects.

7.2 **CONCEPTUAL VIEW**

Conceptual view section describes what software architecture is used for the current system, and how it is represented. Required functionalities are collected from different CASPAR project documents and CASPAR meetings. A technological context respect to SWKM components as well as to the rest of the CASPAR system is presented.

7.2.1 **Responsibilities and goals**

The following responsibilities can be identified for Finding Aids [D1301]:

- Support the user application in locating the relevant data.
- Search AIPs, based on conditions upon the Descriptive Information.
- Function as the "link" between the end-user (consumer or digital archive) and the rest of the CASPAR system, with respect to the search and retrieval facilities.
- Persistence of Descriptive Information

7.2.2 **Technology context**

The system of archiving this responsibilities and Goals (Section 7.2.1) is by using stable technologies, as Web Services, RDF, etc., but this do not limit the scope of the overall Finding Aids architecture which is to stands the base for a long term preservation of DescInfo and related AIPs.

Figure 3 shows the technological approach employed on the Finding Aids that involves three different levels: (a) Data layer, (b) Business layer, and (c) User Interface layer.

---

**Figure 2 – Finding Aids in Strawman Architecture**

In concrete, Finding Aids is a component that allows users to search for CASPAR AIPs based on the associated Descriptive Information objects.
At a certain time a Finding Aids CASPAR component is deployed and configured to interact with the rest of a CASPAR installation, i.e. the minimum set of CASPAR components needed for digital preservation.

Test bed (i.e. artistic) partners provide digital objects and appropriate CASPAR tools (RepInfo Toolbox) aids Designated Community on creating appropriate Representation Information and the other necessary OAIS related information. Archival Information Packages (AIPs) are created and stored on CASPAR system. By interacting with Finding Aids test bed partners associate specific Descriptive Information to created AIPs.

Finding Aids logic allows storing DescInfo objects, the related AIP Identifier, and other Data relevant for preservation purposes. One of the scheduled implementations of this architecture uses Semantic Web Knowledge Manager (SWKM) to preserve internal data structures as well as DescInfo object structure (graphspaces) and documents (namespaces).

![Diagram of Finding Aids technological diagram](image)

**Figure 3 – Finding Aids technological diagram (example with SWKM)**

The Finding Aids support the user application in locating the relevant data. The user (application) is responsible to provide access to Finding Aids overall interface (see Section 7.2.4).

Consumer wants to search AIPs based on conditions upon the Descriptive Information, s/he specifies a certain Query (an RQL) and the Finding Aids answer with the corresponding Result Set based on stored DescInfo information and associated AIPs.

### 7.2.3 High level finding aids

The CASPAR Finding Aids is a software component that provides the functionality for the discovery of Archival Information Packages (AIPs) according to the OAIS Reference Model. From the above Figure 3 FA it is represented by the Finding Manager and the Finding Aids and an implementation specific’ data store.

In order to achieve its goal, the FA is based on two basic components:

1. Finding Registry, and
2. Finding Manager.
A Finding Registry supports the publication and discovery of Finding Managers; in the same way a UDDI server supports the publication and discovery of Web Services. In CASPAR, the concept of Finding Registry is introduced in order to be independent from any specific technology or de facto standard. Clearly, UDDI is a source of inspiration for Finding Registry and any UDDI implementation is a possible choice for implementing the Finding Registry; yet it is important to maintain these two notions separate and independent from each other.

In a CASPAR installation, there can be several Finding Registries, cooperating with each other in order to carry out their overall goal, possibly in a way that is transparent to applications; that is, an application needs not, and in general will not be aware of the existence of many Finding Registries that cooperate with the one the application interacts with.

Finding Registries follow authentication and authorization mechanisms as the rest of CASPAR components. This way just users with appropriate roles will get access to Finding Aids functionality, e.g. look for AIP ID from querying DescInfo stored objects, store / update DescInfo objects, discovery of Finding components.

A Finding Manager supports the management of Descriptive Information (DescInfo, for short), and is bound to a language for defining and for querying DescInfo. For example, a Finding Manager may talk SQL, another one RDF/S, another one XML.

![Figure 4 - Finding Registries and Finding Managers](image)

Every Finding Manager registers with at least a Finding Registry in order to be discovered by applications. The following UML class diagram illustrates the relations amongst the entities introduced so far within a CASPAR installation.

Every CASPAR installation has at least a Finding Registry, while a Finding Registry can only serve a single installation. A Finding Registry registers at least one Finding Manager, while a Finding Manager may be registered with more than one Registry for reliability reasons. Every Finding Manager has exactly one Language, while several managers can talk the same language. Finally, every Manager manages exactly one DescInfo Repository (set of schema elements and associated objects), and vice versa a DescInfo Repository belongs to exactly one Manager.

### 7.2.4 Finding aids models overall architecture

From a functional point of view, a Finding Registry supports three main functionalities:

1. Synchronization with the other Finding Registries of the same installation.
2. Registration and Deregistration of a Finding Manager.

A Finding Manager registers by providing a description of itself to the Registry. This description (FMDesc) contains required information, such as:

- (Data definition & Query) language spoken by the Finding Manager.
• Handle for invoking the Finding Manager.

Additionally, an FMDesc object can contain information concerning properties of the Finding Manager that applications consider useful for discovery purposes.

From a functional point of view, a Finding Manager supports three main functionalities by using a Data Manipulation Language:

1. Management of DescInfo:
   a. At the schema level: create, delete and browse DescInfo schema elements (i.e., tables or classes or DDTs).
   b. At the object level: create, delete, update and browse DescInfo objects (i.e., tuples or objects or documents).

2. Management of the association between DescInfo objects and AIP identifiers, including usage of these associations for AIP discovery:
   a. Create, delete query and browse association instances (i.e. (AIP-id, DescInfo-id) pairs).
   b. Discovery of AIPs via queries on DescInfo objects.

3. Primitives for the preservation of DescInfo, at all levels. These are: export and import of all of the above: DescInfo schema elements, objects and associations with AIPs.

A user application interacts with the Finding Manager using a particular Query Language (QL) for querying and/or browsing the DescInfo Repository.

Below, Figure 5 sketch relations between above-mentioned Data Definition and Query languages, Descriptive Information schema and object, and Archival Information Packages.

![Figure 5 – Finding Aids conceptual components model](image)

Finding Aids components, Finding Manager and Finding Registry, offer three main interfaces to satisfy all required functionality.

Finding Registry access point is called ‘discoveryFM’, which is divided into two stereotypes: (a) Access that allows register, unregister, and search Finding Manager components; and (b) Manager that allows synchronization between different Finding Registries. See Figure 6 - Finding Aids overall interface, <<Interface>> FindingRegistry for methods signature.

Finding Manager access points are called ‘storeDescInfo’ and ‘discoveryAIP’ these and are divided into five stereotypes:

- **storeDescInfo** methods signatures are grouped into following stereotypes:
  - DescInfo schema management stereotype
  - DescInfo object management stereotype
The two remaining stereotypes from Figure 2 show FM initialization methods signature, i.e. to set the Data Definition Language and the Query Language.

**Figure 6 - Finding Aids overall interface**

### 7.3 LOGICAL VIEW

Logical view section presents the class design model and significant sequence diagrams. Finding Aids overall interfaces is decomposed into subsystems and packages. And for each package, its decomposition into classes and class interactions. Classes are described according to their responsibilities.
The Finding Manager component is in charge to interact with SWKM by using two SWKM basic services: IMPOR TER (Section Error! Reference source not found.) and QUERY (Section Error! Reference source not found.) services, it is described in next sections.

7.3.1 Finding manager interfaces

Four main abstract classes form Finding Manager logical component:

- FindingManager abstract class is responsible for FM configuration data and for parse queries. It is also in charge of providing access to the Result Set.
- DescInfoSchemaManager abstract class provides the functionality to create, delete, and list DescInfo stored schemas. Uses SWKM to store DescInfo schemas (RDFS) by calling the store method.
- DescInfoObjectManager abstract class manages the DescInfo object itself, which includes the association with the corresponding AIP. DescInfo objects are stored on SWKM by calling the store method.
- DescInfoRepository abstract class is responsible for the FM storage. Three main concepts are stored: (a) DescInfo schemas, (b) DescInfo objects, (c) specific FM information, including AIP objects¹ and their association with DescInfo Objects.

![Diagram of Finding Manager Logical View]

Figure 7 - Finding Manager Logical View

To instantiate a concrete Finding Manager designer should implement it by extending described abstract classes. To do so a concrete Data Definition Language and a concrete Query language need to be chosen. So, for the SWKM instantiation of Finding Manager the Data Definition Language is "RDF/S" and the Query Language is "RQL".

¹ In Finding Aids context an AIP object is not the true AIP but a representation of it that allow to identify the true AIP, i.e. its ID.
7.3.2 Finding Manager class diagram

This section describes how the previous introduced interface is instantiated on a concrete Finding Manager that uses Web Services to accept client calls and uses SWKM and Relational DB on the Data layer (Figure 3 – Finding Aids technological diagram (example with SWKM)).

Figure 8 - Finding Manager with SWKM shows how SWKM (ClientFactory class) is integrated on the concrete Finding Manager from FM Logical View (Figure 7 - Finding Manager Logical View) as well as how all FM interfaces are implemented (light blue classes) to support SWKM Data and Query languages.

7.3.3 Sequence diagrams examples and source code example

Two main sequence diagrams are presented: the first is the sequence corresponding to the storage of a concrete DescInfo (schema and object), and the second is to perform a RQL query.

Finding Manager contains a component that is as SWKM client and it is initialized as follows:

```java
DbSettings db = new DbSettings();
db.setDbName( dbName );
db.setUsername( dbUsr );
db.setPassword( dbPsw );
db.setHost( dbHost );
db.setPort( dbPort );
db.setProtocol( dbProtocol );
db.setRepresentation( DbRepresentation.HYBRID );
swkmClient = ClientFactory.newClientToWebApp ( swkmHost, swkmPort, swkmContext ).with(db);
```
Storing a new DescInfo schema and DescInfo object is performed by three main steps:

- **DescInfo schemaID**: create a new schema for that object or get the schema ID of a previously stored schema.

  ```java
  Collection<RdfDocument> col = new ArrayList<RdfDocument>();
  RdfDocument rdfdoc = new RdfDocument(
      (String)descInfoSchema.getSchemaID().getID(),
      (String)descInfoSchema.getSchema(),
      (Format)(descInfoSchema).getSchemaFormatSWKM() );
  col.add ( rdfdoc );
  fm_.swkmClient_.importer().store(col, Deps.WITHOUT);
  ```

- **DescInfo object**: create a new descriptive information object based on information provided as well as AIP object used from FM point of view. Associate these objects.

  ```java
  Collection<RdfDocument> col = new ArrayList<RdfDocument>();
  RdfDocument rdfdoc = new RdfDocument(
      ((DescInfoObjectImpl)descInfoObj).getDescInfoRoot().getDIRoot(),
      (String)descInfoObj.getDescInfoObj(),
      (Format)(descInfoObj).getDescInfoFormatSWKM() );
  col.add ( rdfdoc );
  fm_.swkmClient_.importer().store(col, Deps.WITH);
  ```

- **Store**: DescInfo object and association with AIP is stored on repository.

See table below:

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4</td>
<td>When a new AIP object is created and a DescInfo is provided using the DescInfoSchemaManager interface also a schema that represents that information is created and stored into the repository. If the schema already exists, the calls to get the schema ID replace these calls.</td>
</tr>
<tr>
<td>5, 6, 7, 8</td>
<td>A new DescInfo object is created with provided information and an ID is assigned to it.</td>
</tr>
<tr>
<td>9, 10</td>
<td>A new (FM) AIP object is created with CASPAR AIP ID.</td>
</tr>
<tr>
<td>11, 12, 13</td>
<td>An association between DescInfo object and AIP object is created and all information stored on repository.</td>
</tr>
<tr>
<td>14</td>
<td>Created DescInfo object ID is returned to caller.</td>
</tr>
</tbody>
</table>
Finding Manager main functionality is help on looking for AIP objects thought DescInfo querying. In order to do so at least two queries on Repository should be made from a logical point of view. One to get related DescInfo objects and other to get those associated AIP IDs. Let's take a closer look on following steps from sequence diagram.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2</td>
<td>A generic user (or application) submits a query by creating it and calling the discoveryAIP method.</td>
</tr>
</tbody>
</table>
| 3, 4 | The query is appropriately parsed if necessary and passed to the RDF Suite to get the related items. SWKM Client call example:  
```java  
rsSWKM = (ResultSetSWKM) discoveryDIOBJECTS ( query );  
```
| 5 | The result set is parsed and relevant items that can be show to caller (DRM functionality) are used to retrieve all associated items (AIP IDs).  
```java  
rsAIP = (ResultSetAIP) descInfoRepositoryMng.  
getAssociatedAIPFromResultSet ( rsSWKM );  
```
| 6 | Result set is created and returned to caller. |
7.4 FINDING MANAGER USE

Finding Manager invocations using web service calls is presented in this section. First the set up of a web client and then the examples of how this web client can be used to persist a DescInfo schema and a DesdInfo object, as well as to discovery AIPs.

In order to do so it is necessary to use the FM stubs classes and libraries. Download FM stubs last version from:

And required libraries from:
http://wiki.casparpreserves.eu/pub/Main/FIND_Use/20080617_FMCclient-libs.zip

Client FM application can set up a client following this example:

```java
public class MyFINDWebClient {
    private FindingAIDSManager find;
    public MyFINDWebClient () {
        URL findURL = new URL("http://domain.com:8080/FIND/FindingManager?wsdl");
        FindingAIDSManagerService findService = ServiceFactory.createService (FindingAIDSManagerService.class, findURL);
        find = findService.getFindingAIDSManagerPort();
    }
}
```

Let's use DescInfo schema from Section 9.2 as simple example of textual Descriptive Information into defined FM web client. It is necessary to create the schema object (DescInfoSchemaImpl class) and call the createDescInfoSchema method:

```java
DescInfoSchemaIDImpl schemaID = new DescInfoSchemaIDImpl ( );
schemaID.setID ( "write_your_schemaID_here" );
```
Once DescInfo schema is stored into a Finding Manager it is possible to persist its instances, the DescInfo objects. First web client application creates a DescInfo object (DescInfoObjectImpl class) and then calls the createDescInfoObject method:

```java
DescInfoObjectRootImpl descInfoRoot = new DescInfoObjectRootImpl ();
// DescInfo Root is the DescInfo identifier that must be into the RDF file also
descInfoRoot.setDIRoot ("write_your_descInfoRoot_here");
DescInfoObjectImpl descInfo = new DescInfoObjectImpl ();
descInfo.setDescInfoRoot (descInfoRoot);
descInfo.setDescInfoID (schemaID);
descInfo.setDescInfoDescription ("write_your_descInfo description here");
descInfo.setDescInfoFormatSWKM (Format.RDF_XML);
descInfo.setDescInfoObj(IOUtils.readFileAsString (new File(descInfo_file_name )));
DescInfoSchemaIDImpl schemaID = new DescInfoSchemaIDImpl ();
schemaID.setID ("write_your_schemaID_here");
find.createDescInfoObject (schemaID, descInfo);
```

After DescInfo objects are persisted on Finding Manager and corresponding associations to AIP Identifiers made, web client applications can use Finding Manager to discover DescInfo objects by performing a RQL query as follows:

```java
Format format = Format.RDF_XML;
String queryStr = "write_your_query_here";
QuerySWKM query  = new QuerySWKM (format, queryStr);
ResultSetAIP resultSet = (ResultSetAIP) find.discoveryAIP (query);
```

For a full description of web client calls (including associations between DescInfo objects and AIPs), data to be used, FM installation, etc. please see Finding Aids wiki page at:
http://wiki.casparpreserves.eu/bin/view/Main/Task1Id2305

8 AUXILIARY TOOLS FOR DATA INPUT AND VISUALIZATION

This section provides guidelines for creating instantiations of ontologies using auxiliary tools (e.g. Protégé Editor) and it describes a tool Transformer for loading them to SWKM.

Subsequently Section 8.2. presents the DC member tool.

Finally, Section 8.3 presents the StarLion visualization tool.
8.1 PROTEGE AND TRANSFORMER

Here we describe how one can use CIDOC CRM and CIDOC FRBR schemas to create instances of them using Protégé. Specifically we describe the process of documenting the musical opera “Avis de tempete” using a schema AVIS.rdfs which is a specialization of CIDOC CRM and FRBR.

Note: If you want to create only instances of FRBR (which is an extension of CIDOC_CRM) you don’t need AVIS.rdfs (you only need CIDOC.rdfs and FRBR.rdfs).

Prerequisites:
- You have installed Protégé - [http://protege.stanford.edu/](http://protege.stanford.edu/)
  (for example, Protégé 3-3 beta)
- You have installed Java 5 - [http://java.sun.com/j2se/1.5.0/](http://java.sun.com/j2se/1.5.0/)

For creating new instances of CRM or FRBR you need the RDF files describing the schema.
For CRM and FRBR you can download it from:
- [http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf](http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf)

The AVIS specific schema you can find in the package.

In order to be opened by Protégé, the schema files (i.e. CIDOC4.3.rdf, FRBR.rdf, AVIS.rdf) should use rdf:about instead of rdf:ID. This transformation can be done by the method

```
transformSchemaForProtege()
```

which is part of `gr.forth.ics.transformer` package.

This method takes as input two String parameters: the name of the initial file (that needs to be transformed) and the name of the new file (that will be created).

Below we describe the process in more detail.

Extract the Transformer.zip archive in a folder locally (let’s name it C:\Transformer). The folder already contains the compiled classes but in case you want to compile again from a command line type:

```
> cd C:\Transformer
> ant
```

To execute the application you type:
```
> ant run
```
A GUI of the application should open. You will navigate using “Browse” in the “Transformer” folder and select the original file desired (for example “CIDOC.rdfs”) and then fill the field “Resulting file” with the name of the new file (for example “CIDOC_protege.rdfs” – you will specify also the rdfs extension !). Then choose option “Transform Schema for using with Protege”. Then click “OK”.

Note: The “Clear” button resets the values for the two fields (original file and resulting file)

After you pressed “OK” you should see a blue message indicating the location where the new file was created.

You should repeat this step also for the FRBR.rdfs file and transform it creating a new rdfs file (let’s name it FRBR_protege.rdfs) and for AVIS.rdfs file (AVIS_protege.rdfs).

Note: From now on we assume that the RDFS files you use are these transformed files.

Having the new rdfs files we need to load them in Protégé. Because Protégé does not accept the importing of multiple rdfs files in a single project we have to create 3 projects, one for each schema and then we have to define the including relation between them.
1. To create the project corresponding to CIDOC:

1.1 File-> New Project
Select RDF Files and check “Create from existing files”. Then press “Next” as shown in Fig.1
Then in the new window (Fig.2) you select the RDFS file, leave blank for the instances file (we don’t have yet an instance file) and introduce an URI for the namespace (http://cidoc.ics.forth.gr/rdfs/CIDOC4.3.rdf) and then click finish.

1.2 You click “Save Project” and a window similar to that from Fig. 3 appears:
In the first field complete the name of the .pprj (Protégé project), in the second you have the name of the rdfs file imported, in the third the name of the file which will store instances of the given schema and in the fourth the URI used for the namespace.

Check the last option and then press “OK”.

2. To create the project corresponding to FRBR schema:
Repeat the steps 1.1 and 1.2 but this time for the FRBR_protege.rdfs file
For namespace complete: http://cidoc.ics.forth.gr/rdfs/FRBR.rdfs

3. To create the project corresponding to AVIS schema:
Repeat the steps 1.1 and 1.2 but this time for the AVIS_protege.rdfs file
For namespace complete: http://cidoc.ics.forth.gr/rdfs/AVIS.rdfs

Note: At the finishing of these steps you should now have 3 Protégé projects: CIDOC_protege.pprj, FRBR_protege.pprj and AVIS_protege.pprj

You should now create the relations between these projects.
Open the FRBR_protege.pprj project and then from the “Project” menu select “Manage included projects” and then add CIDOC_protege.pprj and then “OK” for two times.
Save this change.

Then open the AVIS_protege.pprj file and import the FRBR_protege.pprj file within it. Save the modification.
Finally you should have a structure as depicted in the next figure.
Exit Protégé and open again the Avis Project.

Note: When opening the project for Avis you can receive from Protégé an error like the one depicted before (see next figure)
This is a bug of Protégé which can be fixed by opening the Protege.lax file and add at the end of the line that starts with `lax.class.path` the entry:

```
plugins/edu.stanford.smi.protegex.rdf_backend/xerces.jar
```

Then save the file and restart Protégé.

To start to create instances of AVIS schema first open the Avis Protégé Project.

A window similar to the one presented in the next Figure should open.
Click tab “Instances” and then in the left pane you select the class you want to instantiate, for example E53.Place, and then in the “Instance browser” click the icon “Create Instance”. The instance editor will be populated with all the properties that we can instantiate (see next figure).
Let's add an instance of the P3F:has_note property by clicking the “Add Value” button and then filling with the desired string value (see next figure)

In this way you can continue to instantiate the schema by adding more class instances and property instances. After you saved the project you will find the result of the instantiation in the specified instance file from the project (for example AVIS_in.rdf).

Note:
We show before how one could instantiate the AVIS schema but the same considerations remain valid if one wants to instantiate the FRBR
ANNEX 1

For the data file generated by Protégé to be processed using SWKM one can use method

\[
\text{transformDataForRDFSuite()}
\]

\[
\text{from gr.forth.ics.transformer package.}
\]

This method removes label elements and also deletes the potential CDATA elements but preserves the content of the CDATA elements.

The method takes as input two String parameters: the name of the initial file (that needs to be transformed) and the name of the new file (that is created).

To execute the application and to create the new rdf file for “Avis de tempete” you type:

> ant run

Select the original file (“avis.rdf” – generated with Protégé; you can find it also in the “Transformer” folder) and specify the name of the new file (for example “avis_transformed.rdf” – do not forget to specify also the rdf extension for this file!)

Select the second option from the list: “Transform data for using with RDFSuite”

NOTE: avis.rdf is the RDF file that describes “Avis de tempete” opera. It was generated using Protégé as described above.

ANNEX 2

Below is presented the listing of the “Avis de tempete” instantiation that was generated by Protégé.

```xml
<?xml version='1.0' encoding='UTF-8'?>
```
&AVIS_rdfs;AVIS_tr_Instance_15
&AVIS_rdfs;AVIS_tr_Instance_6
&AVIS_rdfs;AVIS_tr_Instance_12
&AVIS_rdfs;AVIS_tr_Instance_3
The composition of the 19 sequences

The development of the electronics
<CIDOC4_3_rdfs:P125F_used_object_of_type rdf:resource="&AVIS_rdfs;AVIS_tr_Instance_23"/>
</CIDOC4_3_rdfs:E65_Creation>
</CIDOC4_3_rdfs:E73_Information_Object rdf:about="&AVIS_rdfs;AVIS_tr_Instance_21"
CIDOC4_3_rdfs:P3F_has_note="The electronics of ADT"
/>
</CIDOC4_3_rdfs:E55_Type rdf:about="&AVIS_rdfs;AVIS_tr_Instance_22"
CIDOC4_3_rdfs:P3F_has_note="AddAn"
/>
</CIDOC4_3_rdfs:E55_Type rdf:about="&AVIS_rdfs;AVIS_tr_Instance_23"
CIDOC4_3_rdfs:P3F_has_note="ResAn"
/>
</CIDOC4_3_rdfs:E55_Type rdf:about="&AVIS_rdfs;AVIS_tr_Instance_24"
CIDOC4_3_rdfs:P3F_has_note="Moby Dick"
/>
</CIDOC4_3_rdfs:E55_Type rdf:about="&AVIS_rdfs;AVIS_tr_Instance_25"
CIDOC4_3_rdfs:P3F_has_note="King Lear"
/>
</CIDOC4_3_rdfs:E55_Type rdf:about="&AVIS_rdfs;AVIS_tr_Instance_26"
CIDOC4_3_rdfs:P3F_has_note="The conception of the sound spatialisation"
>
</CIDOC4_3_rdfs:P12F_occurred_in_the_presence_of rdf:resource="&AVIS_rdfs;AVIS_tr_Instance_27"/>
</CIDOC4_3_rdfs:P92F_brought_into_existence rdf:resource="&AVIS_rdfs;AVIS_tr_Instance_27"/>
</CIDOC4_3_rdfs:P94F_has_created rdf:resource="&AVIS_rdfs;AVIS_tr_Instance_27"/>
</CIDOC4_3_rdfs:E65_Creation>
</CIDOC4_3_rdfs:E29_Design_or_Procedure rdf:about="&AVIS_rdfs;AVIS_tr_Instance_27"
CIDOC4_3_rdfs:P3F_has_note="Positioning loudspeakers"
/>
</CIDOC4_3_rdfs:E84_Information_Carrier rdf:about="&AVIS_rdfs;AVIS_tr_Instance_28"
CIDOC4_3_rdfs:P3F_has_note="The file on the MUSTICA Server"
>
</CIDOC4_3_rdfs:P128F_carries rdf:resource="&AVIS_rdfs;AVIS_tr_Instance_27"/>
</CIDOC4_3_rdfs:E84_Information_Carrier>
</CIDOC4_3_rdfs:E29_Design_or_Procedure rdf:about="&AVIS_rdfs;AVIS_tr_Instance_29"
CIDOC4_3_rdfs:P3F_has_note="The partition"
>
</CIDOC4_3_rdfs:P106F_is_composed_of rdf:resource="&AVIS_rdfs;AVIS_tr_Instance_30"/>
</CIDOC4_3_rdfs:E29_Design_or_Procedure>
</CIDOC4_3_rdfs:E82_Actor_Appellation rdf:about="&AVIS_rdfs;AVIS_tr_Instance_3"
CIDOC4_3_rdfs:P3F_has_note="Georges Aperghis"
/>
</CIDOC4_3_rdfs:E29_Design_or_Procedure rdf:about="&AVIS_rdfs;AVIS_tr_Instance_30"
CIDOC4_3_rdfs:P3F_has_note="The flute part"
>
<AVIS_rdfs:PX2_employs_profile rdf:resource="&AVIS_rdfs;AVIS_tr_Instance_32"/>
The libretto, written by Georges Aperghis and Peter Szendy. This opera is "Avis de Tempete" is an opera by Georges Aperghis for ensemble and electronics. The libretto, written by Georges Aperghis and Peter Szendy. This opera is
mainly inspired by Melville's Moby Dick and Shakespeare's King Lear. The electronics in "Avis de tempete" consists in nineteen sequences. These sequences have been created before the instrumental composition, which is based on this electronic material.

8.2 DC MEMBER TOOL

8.2.1 What is the DC member tool?

DC member tool is a web application developed by the CNRS team within the CASPAR project. It comes from one key idea of the artistic testbed: for the artistic partners, preservation is first a question of access to, manipulation of and enrichment of archives by the Designated Community (DC). In other terms, the DC members are those that can make the archives live and so give sense to their preservation. However, DC members are not necessarily computer scientists neither CASPAR experts, therefore an interface must be provided to them in order to foster them to play their role of preservation engine. This is a way to emphasize that such a tool must fit with the skills and limits of the performing arts people that are not specialists of Knowledge Management.

In the case of performing arts, the DC is the set of people who can preserve a work by trying to reperform it. We consider that the DC expectation of re-performance of a work includes the two other expectations identified in our work: 1/ producing other contents from the reference works; 2/ conforming works for an edition. Regarding the specificities of the testbed four partners, we can say that a DC is at least composed by authors/artists and their assistants, domain experts or critics, and editors.

From a conceptual point of view, a DC member tool for preservation is a set of complementary modes of representation (see figure 24). Each mode of representation is a peculiar way to reorganise, to filter, to display and enrich the archived information about a work. In certain circumstances, a mode may also enable an edition of what is so accessible. Every mode must be designed respectfully to:

- What alone this mode allows to do
- How it is combined with other modes, for their respective purposes
- But most of all how it is integrated with the complete aim of the tool
Indeed the use of the whole set of modes must of course enable the preservation of all the data about a work, namely the preservation of the possibility to reperform this work with authenticity.

The full functional specifications of the DC member tool are available on the CASPAR Wiki. There are two studies to be considered as reference for the understanding of the tool specification:

- First the study we made in 2007 about the user interface within the contemporary arts testbed; the report has been approved by IRCAM, INA, CIANT and UNIVLEEDS (see ‘caspar_4300_user_interface_report’ on the CASPAR Wiki).
- Then the conceptual model of this tool that we have developed in another document (see ‘caspar_4300_DCmemberTool_conceptualmodel’ on the CASPAR Wiki).

These documents provide a complete picture of what is covered by the DC member tool. In the following sections we will focus on the life-cycle representation mode, which is one Archiving oriented mode (see figure 24), as it is what currently bridges a local representation of a work with its description in a standardized ontological language.

### 8.2.2 The life-cycle mode of representation

The study of the specificities of the artistic partners, namely of their funds and the related requirements for their preservation, has led us to consider that a life-cycle representation is a relevant way to face the challenge of the preservation of performing works. By “life-cycle representation” we understand the representation of the whole information produced from the original and virtual conception, to the multiple actualizations that constitute the performances. The DC member tool that we specify assumes then that the general process of archiving must use first, graphically and logically speaking, a life-cycle representation. As a synoptic view over the gathered information about a work, this mode may help the user evaluate the completeness of the archive. The life-cycle approach could be considered as a workflow approach. But, to be more precise, we can say that this approach is more like a specific data flow. It represents and organizes all information gathered for the preservation process. Then, thanks to a filter function, partial views of the archive can be generated to retrieve PDI information like Context and Provenance. Figure 25 shows how this mode is presented in the tool.
as one can see, the DC member tool can be used to produce ontological representation of works. It means that elements, relations and types must be manipulated. Since the DC members may be totally ignorant about the technical features of the knowledge management and about ontologies, the tool must present the peculiar DC vocabulary instead of the exact names of the elements, relations and types of the used ontologies. We call the peculiar DC vocabularies: “specific terminologies” (see figure 26 for an example). However, even if the DC member won’t directly use an expert ontology, there will be one working in background. A matching has to be defined between each specific terminology of the institution and the used ontology.

As one can see, the DC member tool can be used to produce ontological representation of works. It means that elements, relations and types must be manipulated. Since the DC members may be totally ignorant about the technical features of the knowledge management and about ontologies, the tool must present the peculiar DC vocabulary instead of the exact names of the elements, relations and types of the used ontologies. We call the peculiar DC vocabularies: “specific terminologies” (see figure 26 for an example). However, even if the DC member won’t directly use an expert ontology, there will be one working in background. A matching has to be defined between each specific terminology of the institution and the used ontology.

The DC member tool enables one to edit life-cycle representations of works in an interactive fashion. One can select the elements to add to the representation of a work directly from the terminology bar (see figure 26). Then the user can:

- Drag-and-drop the element to desired place on the graphical representation,
- Link it with other elements by drawing relations between them,
- Specify its attributes (name, type, etc.), rights and authenticity information using the element inspector of the GUI (right part of the screen on figure 25),
- Discuss with other people about this element or comment it by clicking on the comment button on the element inspector part of the GUI (this will launch a forum thread created automatically for the specific selected element),
- Delete it using the delete button on the element inspector part of the GUI.

At any stage, the user can save the life-cycle by selecting the Archive\>Save function from the application menu. This will store the current work using a local XML format (see figure 27). Previously saved life-cycles can be loaded into the application using the Archive\>Open/Search function of the application menu.

```
<xml version='1.0' encoding='utf-8' ?>
<graphicalModelData archiveId='archive1206910091332'

termDef='ins'

archiveAuthor='Nicolas Esposito'
archivePlace=''
archiveGuidance=''
archiveDate='31/3/2008'>
<relation id='rnx1206911056124'
type='Was used to create'
breakPosX='null'
guidance=''
anchor='element1206910404820'
target='element1206911041012'/></relation>
<relation id='rnx1206911036150'
type='Was commissioned by'
breakPosX='null'
guidance=''
anchor='element1206910404820'
target='element1206911025414'/></relation>
<relation id='rnx1206911007627'
type='Was documented in'
breakPosX='null'
guidance=''
anchor='element1206910404820'
</xml>
```

Figure 14: XML excerpt of a saved life-cycle

The user can also use templates to speed up the production of life-cycles. Indeed, working with our artistic partners we noticed that there were some similarities between descriptions of different works. Moreover, the use of a template for a life-cycle description can be a best practice within a given institution as it reifies how the institution expect their archives to be described and how complete the description must be. In order to produce a template, the user just have to select the Archive\>Save As Template function in the application menu. The application will then automatically open the new template so that the user can specify which are the default values for the elements and relations in the template and also to provide guidance information about how the template should be filled (the large textbox within the inspector panel on figure 28). Once the template is ready, the user can create a new life-cycle representation based on it by selecting the Archive\>New Archive From Template function from the application menu. This will create a new archive in which all default values will be shown in bold red as long as the user does not change them and in which guidance information will be available on the inspector panel (as read only text) and as tooltip text whenever the user moves the mouser cursor over an element or relation.
8.2.3 From a specific terminology to a standard RDF description

DC member tool was developed keeping flexibility in mind. As a matter of fact, this was a functional requirement as the tool has to suit different institutions within the project and amongst the future adopters of CASPAR technologies. Therefore, it is really easy to customize the application for its deployment within a specific institution. The main step consists in the definition of the terminology of the aimed institution. One has to define which are the elements and the relations that are required to produce a life-cycle description of works produced by the institution. Then, these are provided to the institution. One has to define which are the elements and the relations that are required to require as the tool has to suit different institutions within the project and amongst the future can be iterative and one can always add or remove elements and/or relations to it. The administrator of the application is responsible for maintaining the terminology in accordance with the DC requirements. The administrator may also define a terminology of reference that is automatically proposed when the archive producer wants to create a new archive or, if required, define different terminologies that the user can select depending on the kind of work or the kind of description that she or he wants to produce.
Terminologies are made to suit the vocabulary of a DC or part of it. In order to interact with other works repositories and with the generic CASPAR framework, one needs to express the descriptions of works built with the application in a more standardized fashion. The DC member tool provides a mapping mechanism which enables this shift from a description masking all complexity of ontologies and standards to a description incorporating them. This is the step that enables interoperability. The mapping mechanism is also designed in a flexible manner. In order to create it, the administrator of the application has to edit the mapping section within the XML file describing a terminology (see figure 16).

Figure 16: Example of a terminology description in XML

Mapping mechanism is also designed in a flexible manner. In order to create it, the administrator of the application has to edit the mapping section within the XML file describing a terminology (see figure 30). Once this is done, the user can generate the export of a life-cycle by selecting the Archive>Export to RDF function in the application menu. Figure 31 shows an example of a RDF export generated automatically from a life-cycle, describing it with the ontology CIDOC CRM.

Figure 17: Example of a mapping template
8.2.4 Interaction with the CASPAR framework

The goal of this section is to clarify how the DC member tool produced by CNRS for the artistic testbed interacts with other components in the framework of the project. The main interoperability issue is how to move from a specific description of a work to one which can be handled by the generic components. This is answered by the mapping to a standardized ontology (such as CIDOC CRM, FRBRoo or any other ontology) presented in the previous section.

There are two ways in which the tool communicates with other components:

- Through MustiCASPAR: The DC member tool was integrated into the MustiCASPAR server developed by IRCAM. It furnishes to MustiCASPAR a RDF description of works which is parsed by the later to extract provenance information about a piece.

- Through direct communication with the SWKM: The RDF description of a work produced by the export function is fed to the SWKM using the `store()` service. This will enable exploration of existing archives and elements composing an archive. Expected benefits are the possibility to suggest in the DC member tool a list of composers and of Musical Assistants dynamically, the possibility to refer to an existing element from a benefits are the possibility to suggest in the DC member tool a list of composers and of Musical Assistants dynamically, the possibility to refer to an existing element from a work-in-progress archive (for instance to illustrate that a work is based upon a previously archived work without re-specifying the latest).

8.2.5 Requirements

The tool is designed as a web application; therefore it can run on a local installation of a web server application or on a distant web server. On both cases, the server must be able to interpret PHP code. The only requirements on the user side is to open the URL where the application is installed using a Firefox web browser. This is due to the fact that the application relies on XUL code for the interface definition, which is a language developed by the Mozilla Foundation and used for instance by Firefox and Thunderbird.
8.3 STARLION

StarLion is a visualization tool (java application) for RDF schemas that is being developed by FORTH-ICS. It can load ontologies expressed in .rdfs files or stored at the FORTH-ICS Semantic Web Knowledge Middleware (http://athena.ics.forth.gr:9090/SWKM/).

Distinctive characteristics of StarLion:
(a) provision of Top-k diagrams for aiding the process of understanding large in size ontologies,
(b) configurable force-directed layout algorithms (appropriate for semantic networks),
(c) support of a semi-automatic layout process (where the user can change node positions, nail down nodes, apply layout algorithms, etc),
(d) star graph-based (with variable radius) exploration mode.

StarLion can load more than one namespaces (the dependencies of an ontology), supports multiple frames and visualization views and offers various other features (like scaling).

Related Publications:
Yannis Tzitzikas, Dimitris Kotzinos and Yannis Theoharis, On Ranking RDF Schema Elements (and its Application in Visualization), Journal of Universal Computer Science (JUCS), Special Issue: Ontologies and Their Applications, Nov 2007

Some indicative screenshots follow
StarLion can connect to a SKWM server. For instance one can visualize the ontology of Gap Manager (see deliverable D2102) by connecting to the working installation of GapManager using the following settings:

```
host: 139.91.183.8
port: 3027
webApp: SwkmMiddlewareWS
dbName: marketak_db
dbUser: swkm
dbPass:
```

The typology of modules (as loaded from the registry) and visualized using StarLion are shown next.
9 APPENDICES

9.1 EXAMPLE OF RDF/XML AND TRIG FILE FORMATS

```xml
<?xml version="1.0"?>
<rdf:RDF xml:lang="en"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:ms="file:/home/karvoun/downloads/demo_examples/metaschema.rdf#"
  xmlns:rdfsuite="http://139.91.183.30:9090/RDF/rdfsuite.rdfs#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:cult="file:/home/karvoun/downloads/demo_examples/culture_baseuri.rdf#"
  xmlns:adm="file:/home/karvoun/downloads/demo_examples/admin_baseuri.rdf#"
  xml:base="file:/home/karvoun/downloads/demo_examples/culture_data_baseuri.rdf#" >

  <ms:RealWorldObject rdf:ID="Artist">  

    </ms:RealWorldObject>

  <rdfsuite:Graph rdf:ID="C2"/>

  ...

  #end of schema
  #start of data

  <cult:Cubist rdf:about="http://www.culture.net/picasso132">
    <cult:paints>
      <cult:Painting rdf:about="http://www.museum.es/guernica">
        <cult:technique>oil on canvas</cult:technique>
        <cult:exhibited>

  The RDF/XML file format
```

The RDF/XML file format
The TRIG file format
9.2 DESCINFO SCHEMA AND DESCINFO OBJECT FILE EXAMPLES

Simple schema for textual DescInfo objects, file: descinfo-txt_schema.rdfs

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <rdfs:Class rdf:ID="DescInfo">
  </rdfs:Class>
  <rdfs:Property rdf:ID="Root">
    <rdfs:domain rdf:resource="#DescInfo"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </rdfs:Property>
  <rdfs:Property rdf:ID="DescriptionInformation">
    <rdfs:domain rdf:resource="#DescInfo"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </rdfs:Property>
</rdf:RDF>
```

Template for DescInfo objects, file: descinfo-txt_instance_template.rdf

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:desc-txt="schema-txt.rdf#">
  <bench:DescInfo rdf:about="write_here_file_name">
    <bench:Root>
      <write_here_descinfo_root_identifier/>
    </bench:Root>
  </bench:DescInfo>
</rdf:RDF>
```