Project no. 033572

CASPAR

Cultural, **A**rtistic and **S**cientific knowledge for **P**reservation, **A**ccess and **R**etrieval

Instrument: Information Society Technologies

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

D2101B: *Associated draft reports of knowledge management architecture and tools*

Part of

- D2101: Prototype OAIS-based infrastructure and *associated draft reports of knowledge management architecture and tools*
- D2102 Prototype of registry-related KM services (S, Leader FORTH)

Document identifier:	**CASPAR-D2101B-RP-0101-1_0**
Submission Date:	**12-02-2008**
Due Date:	**15-02-2008**
Work package:	**2100**
Partners:	**All Partners**
WP Lead Partner:	**Task (2103) Leader: FORTH-ICS**
Document status	**FINAL**

Abstract:

CASPAR aims at preserving not only the bits of digital objects but also the information and knowledge that is encoded in these objects. However it is hard to define explicitly what information or what knowledge is. It is therefore very difficult for someone to claim that a particular approach, methodology or technique can indeed preserve information and knowledge. To tackle this issue and for preserving the meaning of digital objects, this document presents a formalization of the notion of intelligibility in a OAIS-compliant manner (it actually extends OAIS) – a result of the work of the first phase of this project. It is worth mentioning that this model has already been published in refereed international conferences and has received encouraging comments. In addition, this document provides more specific and concrete examples of that formalization and provides guidelines, methodologies and component specifications. Along these lines, it also contains the specification of the KM component, a description of the functionality of each planned iteration and a number of testing/validation scenarios. Finally it provides technical details regarding possible installations and deployments of the KM component.

Delivery Type	Internal Report
Author(s)	Yannis Tzitzikas, Vassilis Christophides, Dimitris Kotzinos, Grigoris Antoniou Contributors: CASPAR Consortium.
Approval	
Summary	
Keyword List	
Availability	☒ PUBLIC/CONFIDENTIAL(with footnote)/PRIVATE

Document Status Sheet

Issue	Date	Comment	Author
0.1	Dec 3, 2006	Draft Structure of document and specification of the basic KM Services	Yannis Tzitzikas (FORTH-ICS), Dimitris Kotzinos (FORTH-ICS), Vassilis Christophides (FORTH-ICS)
0.2.	June 3, 2007	Content added	Yannis Tzitzikas
0.3	July 5, 2007	First readable version	Yannis Tzitzikas
0.4	July 17, 2007	Extensions based on the Oxford meeting (July 2007)	Yannis Tzitzikas, Dimitris Andreou
0.5	July 30, 2007	Some revisions, plus some first comments about the implementation phases.	Yannis Tzitzikas, Dimitris Andreou and other members of FORTH.
0.7	Aug. 1, 2007	Changed Document Template plus some minor changes	Sorin Ciolofan
0.8	Oct 1, 2007	Description of DC-aware AIPs and DIPs. Changes based on email discussions in September regarding Virtualization and POM). Updated Class Diagram of Gap Manager. Detailed Specification of Basic SWKM Services (from the prototype description document) Addition of link to the document of Eric Gebers (INA's RepInfo dependencies).	Yannis Tzitzikas

<table>
<tr><td>0.9</td><td>Nov 1, 2007</td><td>Extra comments explaining why the Basic SWKM Services have been designed with intelligibility in mind. (section 7.5).

Addition of Sequence Diagrams (regarding the communication between RepInfoRegistry, POM and CKM).

Addition of a proposed deployment diagram.</td><td>Yannis Tzitzikas</td></tr>
<tr><td>1.0</td><td>Nov 14</td><td>An abstract was added and some sections were restructured.

More specific testing conditions (for the outcomes of each of the planned iterations) are given.</td><td>Yannis Tzitzikas</td></tr>
<tr><td>1.1</td><td>Dec 12</td><td>More detailed specification of the getRelatedConcepts (needed by the Virtualization component) and of the descriptive metadata services.</td><td>Yannis Tzitzikas</td></tr>
<tr><td>1.2</td><td>Feb 12</td><td>Updated descriptive metadata services (addition of the browsing service) needed for the Virtualization component.

Description of the core ontology for exchanging modules, dependencies and DC profiles.

Description of the implementation status of Gap Manager</td><td>All members of FORTH (including Yannis Marketakis)</td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
</table>

Project information

Project acronym:	**CASPAR**
Project full title:	**Cultural, Artistic and Scientific knowledge for Preservation, Access and Retrieval**
Proposal/Contract no.:	**IST-2006-033572**

Project Officer: Carlos Oliveira

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

Project Co-ordinator: David Giaretta

Address:	STFC (formerly CCLRC), Rutherford Appleton Laboratory Chilton, Didcot, Oxon OX11 0QX, UK
Phone:	+44 1235 446235
Fax:	+44 1235 446362
Mobile:	+44 (0) 7770326304
E-mail:	d.l.giaretta@rl.ac.uk

CONTENT

1. INTRODUCTION ... 8

1.1 PURPOSE OF THIS DOCUMENT ... 8
1.2 HOW TO READ THIS DOCUMENT ... 8
1.3 APPLICABLE DOCUMENTS AND REFERENCE DOCUMENTS ... 8
1.4 GLOSSARY ... 9

2 TOWARDS FORMALIZING INTELLIGIBILITY ... 13

2.1 MOTIVATION ... 13
2.2 FORMALIZING INTELLIGIBILITY USING MODULES AND DEPENDENCIES ... 13
2.3 EXTENDING THE INFORMATION MODEL OF OAIS ... 19
2.4 MIGRATION AND EMULATION ... 19
2.5 SOME DIFFICULTIES AND LIMITATIONS ... 20
2.6 INTELLIGIBILITY-AWARE PROCESSES ... 21
2.7 SUMMARY ... 22

3 HOW (OR DIFFERENT WAYS) TO REPRESENT DC KNOWLEDGE ... 23

4 ARCHITECTURE OF SEMANTIC WEB MODELS ... 26

4.1 THE BENEFITS OF ADOPTING SEMANTIC WEB TECHNOLOGIES ... 26
4.2 CORE ONTOLOGY FOR OAIS AND INTELLIGIBILITY-RELATED TASKS ... 26
4.2.1 The Core Ontology for Exchanging Modules, Dependencies and DC Profiles ... 27
4.3 ARCHITECTURAL GUIDELINES ... 28
4.4 MODELING GUIDELINES ... 29
4.5 OAIS AND CIDOC CRM (PROVENANCE AND CONTEXT) ... 31
4.6 KNOWLEDGE MANAGEMENT AND CASPAR TESTBEDS ... 32
4.7 ABSTRACTING FROM SPECIFIC KNOWLEDGE REPRESENTATION LANGUAGES ... 32

5 GENERAL ARCHITECTURE OF KNOWLEDGE MANAGEMENT SERVICES ... 33

6 HIGH LEVEL SERVICES (ANALYTIC DESCRIPTION) ... 37

6.1 DC PROFILE MANAGER INTERFACE ... 37
6.2 REPINFO GAPMANAGER INTERFACE ... 38
6.3 DESCRIPTIVE METADATA SW MANAGER INTERFACE ... 42
6.3.1 Browsing Service ... 42

7 BASIC SWKM SERVICES (ANALYTIC DESCRIPTION) ... 48

7.1 PURPOSE AND SCOPE ... 48
7.2 BACKGROUND: RDF/S NAMESPACES AND GRAPHSPACES ... 48
7.3 SW KNOWLEDGE REPOSITORY (PERSISTENCE, VALIDATION, QUERY, UPDATE) ... 48
7.4 SW QUERY AND UPDATE LANGUAGES ... 48
7.5 BASIC SWKM SERVICES ... 49
7.6 IMPORTER SERVICE ... 50
7.6.1 Store ... 50
7.6.2 Store with Dependencies ... 51
7.7 EXPORTER SERVICE ... 52
7.7.1 Fetch ... 52
7.7.2 Fetch with Dependencies ... 52
7.7.3 Fetch with data ... 53
7.7.4 Fetch with data and dependencies ... 54
7.8 QUERY SERVICE ... 54
7.9 UPDATE SERVICE ... 55
7.10 EXAMPLE OF KNOWLEDGE EVOLUTION ... 56

8 CASPAR KNOWLEDGE MANAGER COMPONENT ... 58

8.1 ITERATIONS 58
8.2 MORE DETAILS ON TESTING 59
8.3 IMPLEMENTATION DETAILS 61
8.4 CASPAR KM INSTALLATION 63
8.4.1 Required software components 63
8.4.2 The SWKM services as a web application 65
8.5 USING THE SWKM SERVICES 67
8.5.1 A SWKM services' sample client 67
8.6 A POSSIBLE DEPLOYMENT 67
8.7 SUPPORTED CHARACTERS IN SEMANTIC WEB MIDDLEWARE MANAGER (SWKM) 68

9 OTHER USEFUL TOOLS 70

9.1 LIST OF TOOLS 70
9.2 USING PROTÉGÉ AND OTHER EDITORS 70

10 EXAMPLE OF RDF/XML AND TRIG FILE FORMATS 71

1. INTRODUCTION

1.1 PURPOSE OF THIS DOCUMENT

CASPAR aims at preserving not only the bits of digital objects but also the information and knowledge that is encoded in these objects. However it is hard to define explicitly what information or what knowledge is. It is therefore very difficult for someone to claim that a particular approach, methodology or technique can indeed preserve information and knowledge. To tackle this issue and for preserving the meaning of digital objects, CASPAR attempts to formalize the notion of intelligibility in a OAIS-compliant manner (it actually extends OAIS) and to provide guidelines, methodologies and components that could aid humans in preserving information and knowledge.

This document is important for all designers and users or the CASPAR components.

1.2 HOW TO READ THIS DOCUMENT

This document summarizes the main 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).

1.3 APPLICABLE DOCUMENTS AND REFERENCE DOCUMENTS

Applicable documents

<<all the previous deliverables of CASPAR>>

[S1] Y. Tzitzikas: Dependency Management for the Preservation of Digital Information. 18th International Conference on Database and Expert Systems Applications, DEXA’2007, Regensburg, Germany, September 2007

[S2] Y. Tzitzikas, G. Flouris: Mind the (Intelligibility) Gap. 11th European Conference on Research and Advanced Technology for Digital Libraries, ECDL’2007, Budapest, Hungary, September 2007

[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), Oberpfaffenhofen/Munich, Germany, October 2007October 2007

Reference documents

[E1] OAIS: Open Archival Information System, International Organization For Standardization, ISO 14721:2003, http://public.ccsds.org/publications/archive/650x0b1.pdf (version of 11 June 2007)

[E2] J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis: Telos: Representing Knowledge about Information Systems, ACM Transactions on Information Systems, 8(4), 1990.

[E3] R.A. Lorie: Long term preservation of digital information, Proceedings of the 1st ACM/IEEE-CS joint conference on Digital Libraries, 346--352, 2001.

1.4 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-nmd.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 poly-hierarchy, which means that a term can 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 TOWARDS FORMALIZING INTELLIGIBILITY

2.1 MOTIVATION

According to the Information Model of the OAIS Reference Model, metadata are distinguished to various broad categories. One very important (for preservation purposes) category of metadata is named *Representation Information* (RI) which aims at enabling the conversion of a collection of bits to something meaningful and useful. In brief, the RI of a digital object should comprise information about the Structure, the Semantics and the needed Algorithms for interpreting and managing a digital object. Figure 2-1 shows the corresponding class diagram.

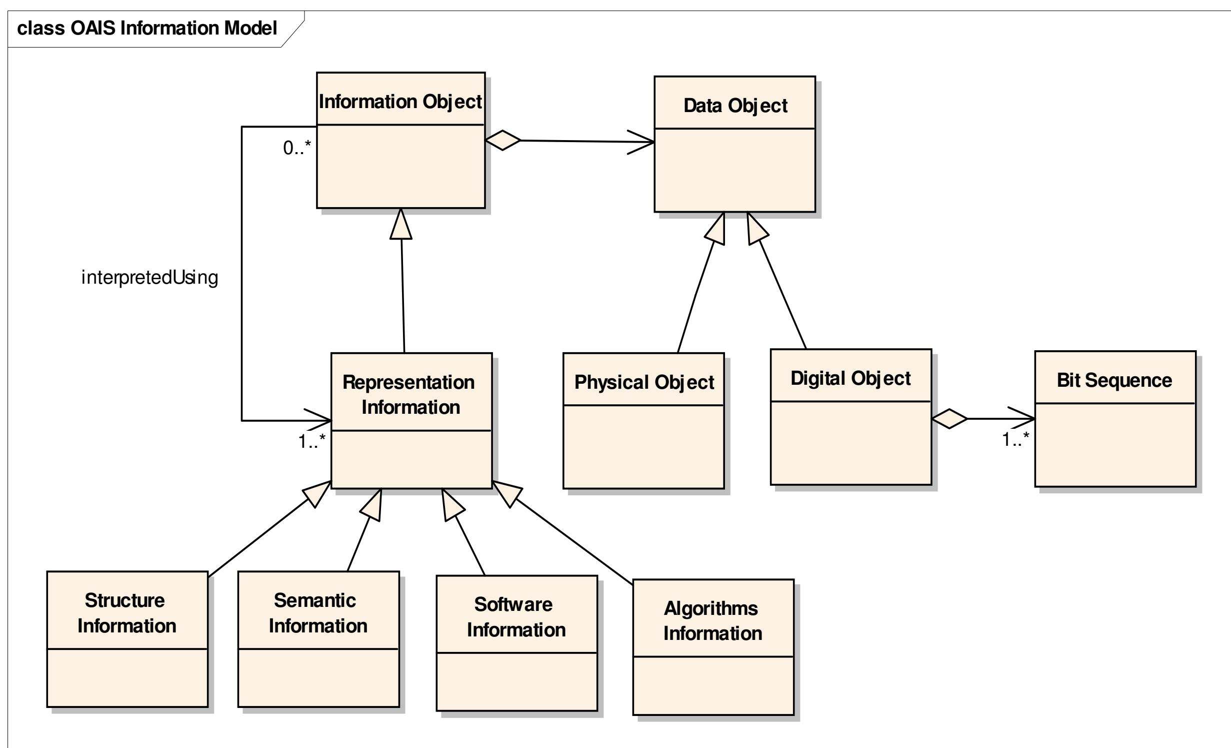


Figure 2-1 The Information Model of OAIS

However for applying OAIS in practice, a number of important questions are raised:

How much representation information should we capture and record?

How this depends on the knowledge of the DC?

How we could model DC knowledge?

What automation could we offer?

2.2 FORMALIZING INTELLIGIBILITY USING MODULES AND DEPENDENCIES

In order to abstract from the various domain-specific and time-varying details, [S1] (i.e. [Tzitzikas, DEXA'2007]) proposed a model for formalizing intelligibility based on the notion of Module and Dependency. A module could be a piece of software/hardware, a knowledge model expressed explicitly and formally (e.g. an Ontology), a knowledge model not expressed explicitly (e.g. Greek Language). The only constraint is that modules need to have a unique identity. Concerning

dependencies, a module t depends on t', written t>t', if t requires t'. Broadly speaking, the meaning of a dependency t > t' is that t cannot function/be understood/managed without the existence of t'. So we model the RI requirements of the OAIS information model as dependencies between modules. Some examples are illustrated in Figure 2-2 For instance, the intelligibility of a digital object README.txt depends on the availability of text editors and knowledge of the English language. The rest examples come from the various testbeds of the CASPAR project (e.g. FITS files are used by astronomers). Dependencies also exist between formally expressed knowledge. For instance, the left part of Figure 2-3 sketches a number of Semantic Web (SW) schemas (recall that a SW schema may reuse or extend elements coming from different schemas) and the right part shows the corresponding dependency graph.

An important question that is now raised is how we could model community knowledge according to the above framework. The knowledge of an actor or community u can be characterized by a ***profile*** Tu that contains those modules that are assumed to be available/known to u (i.e. Tu $\subseteq$T). For example, if u is an artificial agent, then Tu may include the software/hardware modules available to it. If u is a human, then Tu may include modules that correspond to implicit or tacit knowledge. Note that if preservation is done for a particular Designated Community (DC), we may call these *DC profiles*. One can easily guess that what the dependencies really are, strongly depends on the DC and on its purposes.

Now we introduce an assumption, namely the *unique module assumption* (UMA), which is very useful for both theoretical and practical reasons. According to UMA each module is uniquely identified by its name and its dependencies are always the same (in practice we may adopt a more relaxed assumption of the form: different modules have different identities).

If T denotes the set of all modules, then the dependency relation > is actually a binary relation over T. We shall use Nr(t) to denote the direct dependencies of t, i.e. Nr(t)={t' | t > t'}. We shall use $>^+$ to denote the transitive closure of >, and $>^*$ to denote the reflexive and transitive closure of >. Analogously, we can define $Nr^+(t)$ = { t' | t $>^+$ t' } and $Nr^*(t)$ = { t' | t $>^*$ t' }.

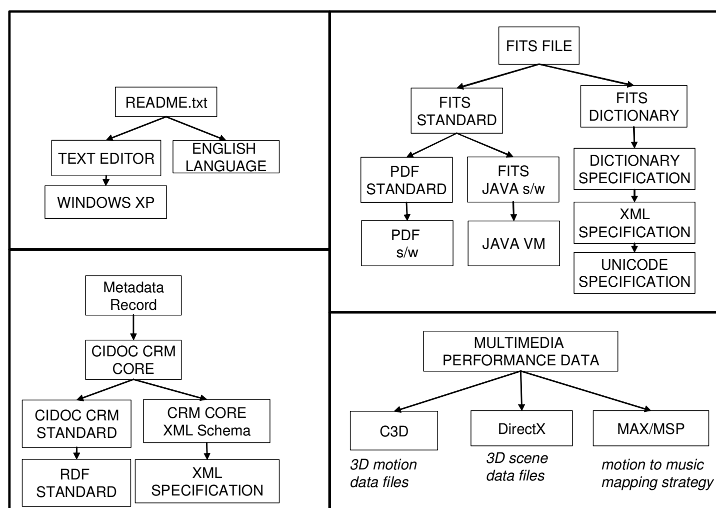


Figure 2-2 Examples of Modules and Dependencies

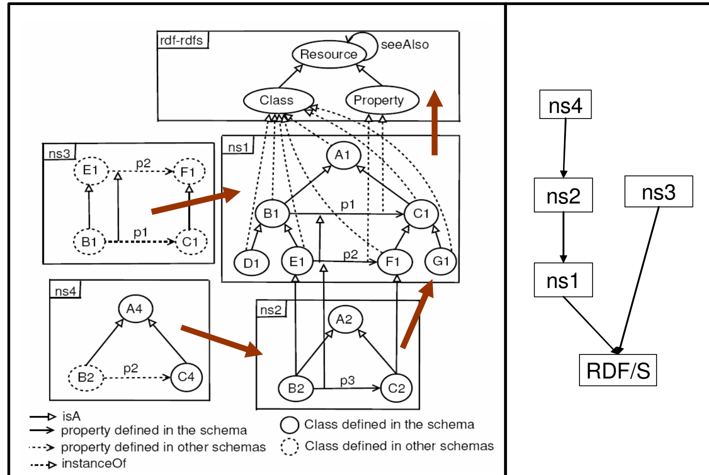


Figure 2-3 Dependencies between RDF Schemas

In order to formalize the notion of intelligibility we introduce the notion of *closure*. The closure of a module t is defined as $C(t) = Nr^*(t)$. The closure of a set of modules S (where $S \subseteq T$) is defined as $C(S) = \cup \{ C(t) \mid t \in S \}$. As Tu is a set of modules, we can define its closure as C(Tu).

Recall that $Nr^+(t)$ is the set of all dependencies of t, i.e. $Nr^+(t)=C(t)-t$. We may denote this by $C^+(t)$, so it is actually the set of all (direct or indirect) dependencies of t. Figure 2-4 shows a dependency graph, and the profile Tu of an actor u.

It follows easily that u can understand a module t if and only if $C^+(t) \subseteq C(Tu)$. In the running example u can understand ty but he cannot understand tx.

We can now define the *intelligibility gap* as the smallest set of extra modules that u needs to have in order to understand a module t. We can denote this by **Gap(t,u)** and it follows easily that $Gap(t,u) = C^+(t)-C(Tu)$ where "-" denotes set difference. In our example $Gap(ty,u)= \emptyset$ while $Gap(tx,u)= \{t1, t2, t4, t5\}$. Notice that due to UMA it is implied that u can also understand t7 and t8 even if they are not elements of Tu. In addition, and due to UMA, we can decide whether a module is intelligible by inspecting only the direct dependencies of t. In particular it holds : $C^+(t) \subseteq C(Tu) \Leftrightarrow max(C^+(t)) \subseteq C(Tu)$. In our example $max(C^+(ty))) = t3 \in C(Tu)$, while $max(C^+(tx))=t1 \notin C(Tu)$.

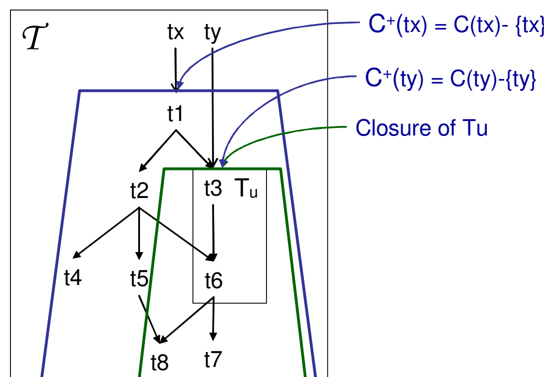


Figure 2-4 Dependency Graph

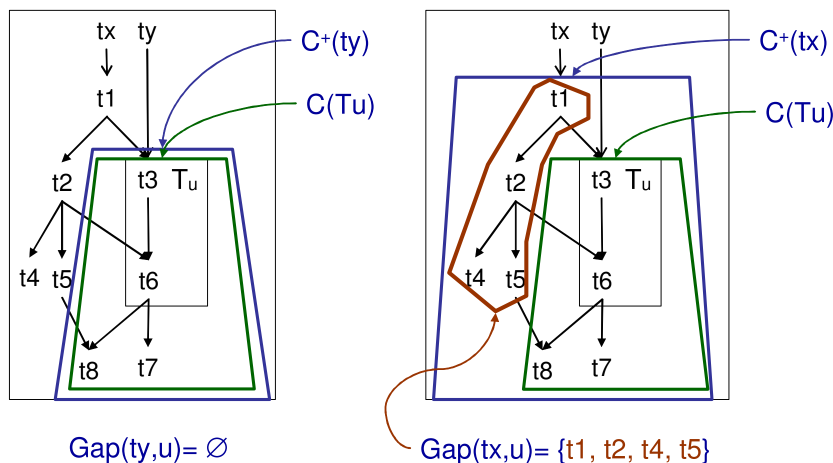


Figure 2-5 Example of Intelligibility Gaps

According to the previous discussion, an intelligibility gap can be filled by getting the missing modules. This means that if we want to preserve a digital object t for a community with profile Tu then we need to get and store only Gap(t,u) plus an id that denotes Tu. Analogously, if we want to deliver an object t to an actor with profile Tu, then the only extra modules that we should deliver to him in order to return him something intelligible, is the set Gap(t,u).

This means that we don't necessarily have to express explicitly the entire community knowledge. We just have to decide what it is assumed to be the DC knowledge and may introduce modules (which could be symbolic values) that denote this knowledge. This approach enforces us to clarify and express what it is assumed to be known (by the designated community) and what is not.

For example if we have to preserve FITS files for astronomers and we make the assumption that this community knows that the FITS standard and the FITS dictionary is, then we do not have to record any extra RI. If on the other hand we want to preserve these files for ordinary users then we would have to record the dependency graph of Figure 2-2.

As another example consider the example of Figure 2-3. Suppose that o=ns4 and that Tu={RDF/S}. In this case we have Gap(o,u)={ns2,ns1}. Suppose that Tu is assigned an identifier denoted by id(Tu), and suppose that id(Tu)=profile3. In this case what we have to store is metadata(o)={ ns2,ns1,profile3}.

Recall that according to OAIS an AIP (Archival Information Package) is actually a format which consist of the Data Object the required RepInfo plus PDI (Preservation Description Information). A DIP (Dissemination Information Package) is an information package delivered to the Consumer in response to an access request and it may differ in form (e.g. TIFF to JPEG) or content (e.g. amount of metadata supplied) to that which resides in the archival store. The adoption DC profiles allows defining the "right" AIPs. Specifically we can define AIPs that that are intelligible for certain communities and at the same time are redundancy free. The same is true for DIPs. This is illustrated in Figure 2-6 where for an object o1 three different AIPs are defined (for DC1, DC2 and DC3). The DIPs of o1 for DC1, DC2 and DC3 are actually the corresponding AIPs without the line that indicates the profile.

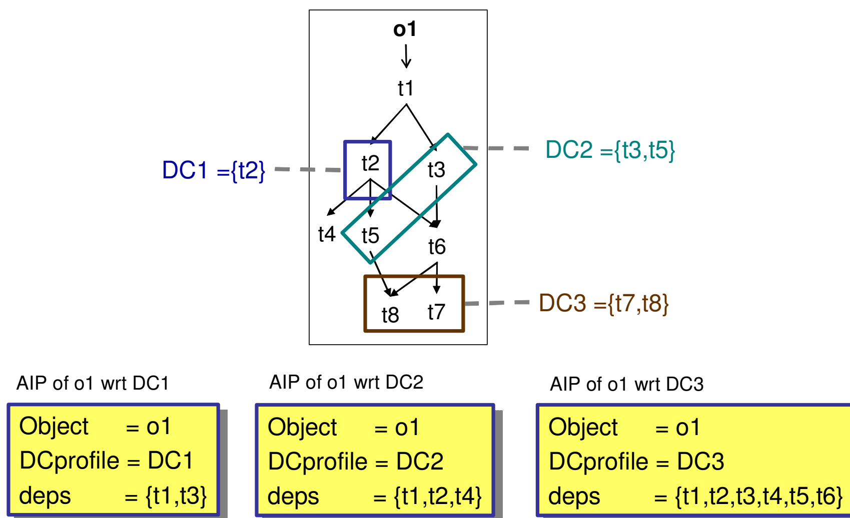


Figure 2-6 Exploiting DC profiles for defining the right AIPs

Of course the less assumptions we make about what the community knowledge is, the more difficult and laborious the problem of recording becomes. With no assumption at all, i.e. if we assume that Tu=∅, then the problem is just impossible.

But what about cross-community interpretability? An important remark is that the more explicitly we have represented the knowledge of communities the more probable cross-community interpretability is. This is illustrated in Figure 2-7. The left part illustrates 2 digital objects a and b each depending on one DC profile A and B respectively. In this case a cannot be understood by community B and b cannot be understood by community A. The right part of the figures illustrates the same situation with the only difference that these two profiles have been analyzed in more detail. In this case gaps can be computed and cross-community intelligibility could be supported. Specifically in that case we have Gap(a,B) = {t1,t3} and Gap(b,A)={t2,t5}.

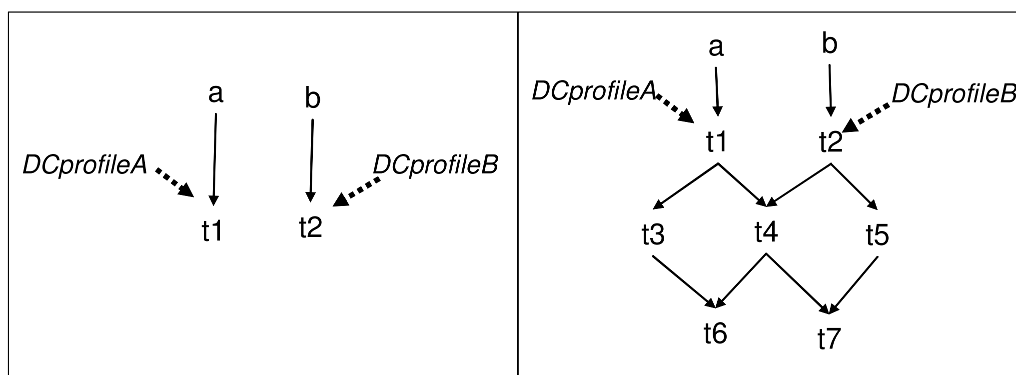


Figure 2-7 Dependencies and Cross-Community Intelligibility

To summarize, the above formalism allowed us to provide specific answers to the following very important questions: (a) how much RI should we create? (b) how this depends on the knowledge of the designated community? (c) what automation can he have?

A Knowledge Manager component could keep stored the dependency graph and the profiles, while a Preservation Data Store could keep the AIPs (as shown in Figure 2-8). The packaging process is responsibility of the Packaging Component. The Knowledge Manager could aid in providing DIPs according to DC Profiles different that those that have been used for archiving the packages.

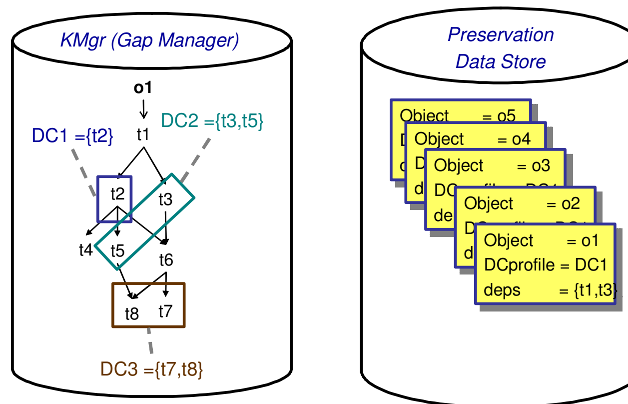


Figure 2-8 Knowledge Manager and Preservation Data Stores

If DC profiles evolve then it would be possible to reconstruct the AIPs according to the latest DC profiles. Such an example is illustrated in Figure 2-9.

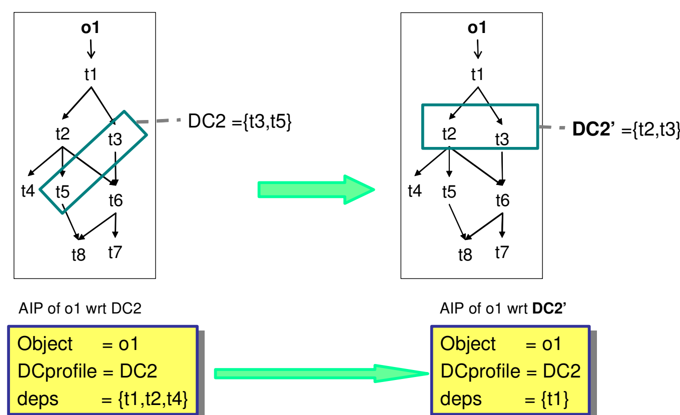


Figure 2-9 Revising AIPs after DC profile changes

2.3 EXTENDING THE INFORMATION MODEL OF OAIS

The above formalization drives to an extension of the Information model of OAIS. In brief the notion of DC Profile should be explicitly represented. A diagram that specifies the extension is shown in Figure 2-10.

Apart from having the additional class Profile and its association with the class Module, this schema suggests having an extensible specialization hierarchy under the class Module. This modelling approach is more flexible than having aggregation hierarchies (as in the original OAIS information model). With aggregation hierarchies one would have to decide and fix at the beginning what kinds of RI there exist. With the proposed approach the kinds of RI may evolve over time. Moreover multiple classification is suggested for the data layer. This means that we may have a module that belongs to more than one subclasses of the class Module. This very useful because in many cases the distinction between structure, algorithms, semantics, etc, is quite blurred.

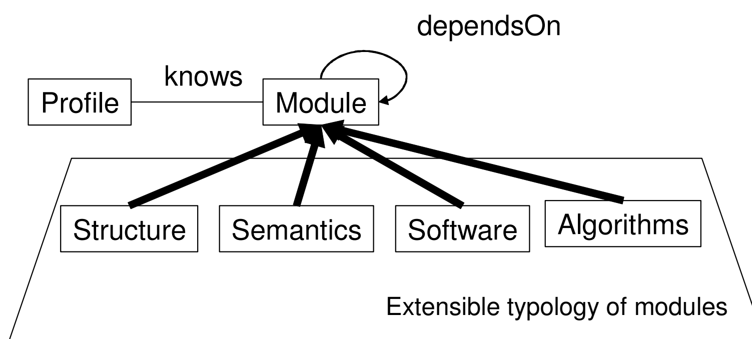


Figure 2-10 Extended OAIS Information Model

The representation of this information model using Semantic Web languages will be discussed in a next section (Figure 4-1).

2.4 MIGRATION AND EMULATION

However, a preservation approach should also be able to tackle the fact that almost everything changes over time. This means that changes in the dependencies should trigger intelligibility checks and may requests for further actions. In addition, it should be remarked that the notion of conversion (or transformation), which is a common practice for achieving interoperability in information systems, could also be exploited for the problem at hand. Specifically, for any module t we can use In(t) to denote the set of all inputs that t can recognize (valid well-formed inputs). For example a module called PDFReader can read all files with type PDF. From this point of view a converter from t to t' can be seen as a function f: In(t) → In(t'). We can model the available converters by a graph C = (T,→) where the edges denote converters. As there may be more than one converters from t to t', the graph is actually a multigraph. In particular, an edge t → t' represents a converter from module t to a module t'. For example, if Nr(o)=t and there is a converter t → t' then if we apply on o the converter t → t', we can get a new digital object o' such that Nr(o') = t'. Suppose an object o and a user u, such that Gap(o,u) ≠ ∅. For understanding o, the user instead of having to find and install the missing required modules, he could apply a number of converters so that to transform o to an object that he can understand with the modules already available to him. Specifically, such a conversion is possible (i.e. C contains the needed converters) if from every element of Gap(o,u) there is a path t →…→ t' in C where t' ∈ Tu (more precisely, t' ∈ C(Tu)). The problem of deciding whether this is possible reduces to the problem of REACHABILITY in directed graphs, i.e. from every t ∈ Gap (o,u) we want to reach at least one element of Tu. The plain REACHABILITY algorithm can return the shortest path (corresponding to the minimum number of needed conversions) for each element in O(|E|) where |E|

the number of the edges. An example of converters is shown in **Figure 2-11** where converters are denoted with dashed arrows. For instance, t1 which is not intelligible by u, can be transformed to an intelligible object if we apply to it the converter t1→ t3, or t1→t6, or t1→t7, or the sequence of converters t1→t2→t7. An alternative formal definition of the notion of conversion and emulation is given in [Tzitzikas & Flouris, ECDL'2007]. However, we should not ignore that converters are themselves modules, therefore we could have a core model like the one shown in Figure 2-12.

We should also point out that such view encompasses preservation approaches that are based on virtualization. For instance, the UVC (Universal Virtual Computer) approach [Lorie, 2001][1] also requires RI (as the specification of a UVC has to recorded and it may change over time). So we could say that the adoption of virtualization approaches just reduces the number of dependencies that we may have (however it does not vanish them). In the same spirit, MDA (Model Driven Architecture - http://www.omg.org/mda/) and OMG standards aim at reducing the dependencies (instead of having a specification of an information system that depends on a number of different technologies, MDA proposes a specification that depends only on OMG standards).

However we have to note that the UVC-approach is not the focus of the CASPAR project.

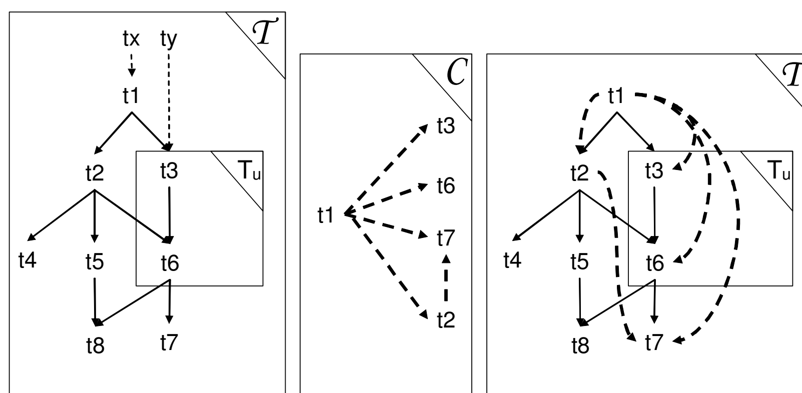


Figure 2-11 Extending the Model with Converters

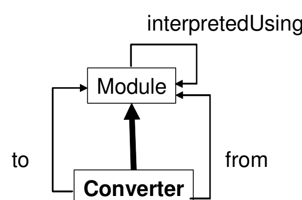


Figure 2-12 Modules and Converters

2.5 SOME DIFFICULTIES AND LIMITATIONS

Let us for example consider the case of Web pages. Consider a digital file named a.html. The extension of the filename gives us a hint about the type of the digital object, so we may write type(a.html)=HTML and and as a.html > HTML, we may generalize and consider that for every object o it holds o > type(o), if type(o) is known. However, an html page is a text that may contain pointers to

[1] R.A. Lorie: Long term preservation of digital information, Proceedings of the 1st ACM/IEEE-CS joint conference on Digital Libraries, 346--352, 2001

other types of data (images, sounds, etc). In order to obtain this content, we need a HTML parser. So we could say to compute the dependencies of a.html we need to have an HTML parser[2].

So in practice we may be unable to compute C(t). We may be able to compute only a part of Nr(t).

2.6 INTELLIGIBILITY-AWARE PROCESSES

Figure 2-13 illustrates the functional model of OAIS. The analysis presented in the previous section suggests that a preservation Information System could adopt the notion of profile in order to support intelligibility-aware services. For instance, it could adopt the following policies: (a) the input (e.g. data objects to be archived) should be intelligible by the system, and (b) the output (e.g. returned answers) should be intelligible by the recipients. The notion of profile could be used as gnomon in these policies.

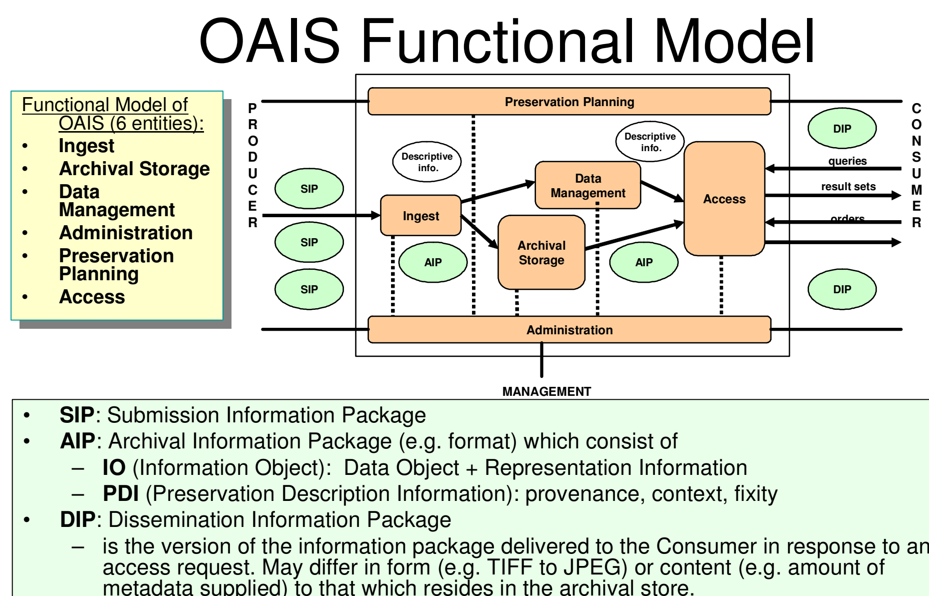


Figure 2-13 The Functional Model of OAIS

Figure 2-14 illustrates some basic steps of these processes. They include the steps of selecting a profile, identifying the gap and filling the gap. Moreover, the impacts of changes have to be identified and the involved parties should be notified. The latter is part of the ongoing curation process. The impacts of changes on the modules and their dependencies are discussed in [Tzitzikas & Flouris, ECDL'2007]. We should remark that these processes correspond to the elements of the functional model of OAIS.

[2] As another example, for a .java named file we need to parse the file in order to extract all import statements, while for a .rdf named file, we need to parse it in order to extract the namespaces it uses.

As remarked previously we may be unable to compute the closure of an object, so we may be unable to compute the intelligibility gap. For this reason a progressive/gradual interaction scheme could be adopted (for more details see [Tzitzikas, DEXA'2007]).

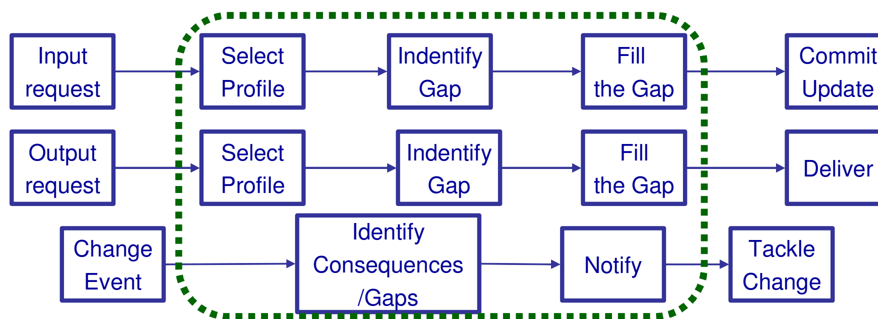


Figure 2-14 Intelligibility-aware Processes

However as they contain intelligibility-related steps, they could be considered **as an extension of the Functional Model of OAIS.**

2.7 SUMMARY

The preservation of intelligibility is an important requirement for digital preservation. In this project we formalized this notion on the basis of modules and dependencies. Recall that dependencies are ubiquitous and dependency management is an important requirement that is subject of research in several (old and new emerged) areas, from software engineering to ontology engineering. Subsequently we formalized the notion of community knowledge in the form of profiles and defined intelligibility and intelligibility gaps. The notion of intelligibility gap is very important as it provides specific answers to questions of the form: what we should be record/deliver to achieve the intelligibility of digital objects by a specific community? Based on these notions we sketched a number of intelligibility-aware processes.

Recall that the preservation of the intelligibility of digital objects requires a generalization (or abstraction) able to capture also non software modules (e.g. explicit or implicit domain knowledge). A modern preservation system should be generic, i.e. able to preserve heterogeneous digital objects which may have different interpretation of the notion of dependency. The dependency relations should be specializable and configurable (e.g. it should be possible to associate different semantics to them). Focus should be given on finding, recording and curation of dependencies. For example, the makefile of an application program is not complete for preservation purposes. The preservation system should also describe the environment in which the application program (and the make file) will run. Recall the four worlds of an information system (Subject World, System World, Usage World, Development World) as identified by [Mylopoulos, 1990][3]. Finally, the provision of notification services for risks of loosing information (e.g. obsolescence detection services) is important.

[3] J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis: Telos: Representing Knowledge about Information Systems, ACM Transactions on Information Systems, 8(4), 1990

3 HOW (OR DIFFERENT WAYS) TO REPRESENT DC KNOWLEDGE

In general it is very hard to formalize and express explicitly the knowledge of a community. However the previous analysis showed that we don't necessarily have to explicitly represent that knowledge: we may just have to agree on symbols that denote that knowledge. However, the more detailed descriptions we have, the more interoperability and cross-community interpretability we can achieve. Unfortunately in most cases there is a lot of tacit and not explicitly expressed knowledge. This is not always because no one dedicated enough effort to represent explicitly a piece of knowledge. We do not have the machinery and the technology to express everything formally. For example, how to represent the knowledge described in a scientific paper whose subject is second-order-logic?

Due to the above reasons, the formal model of intelligibility that was introduced earlier adopted a very broad definition of what a module can be. A module can be an artifact (e.g. a PDF file describing a data format, or a metadata record instantiating an ontology) or just a symbolic value that denotes something very broad, e.g. the notion of "GreekLanguage". This approach can offer flexibility, i.e. ability to encompass different kinds (in terms of specificity and formalization) of modules. This is a quite pragmatic approach.

Despite the aforementioned problems, in some cases there are artifacts that aim at representing the common conceptualization and the consensual knowledge of a community. There are several forms that such artifacts may have. Some key distinctions are available at the Glossary (at the beginning of this document).

The degree of application and usage should be one important criterion for selecting the appropriate models in case more than one exist, because by definition such models should express consensual knowledge. The definition from scratch is a laborious and not very promising approach. It is better adopting or extending appropriately existing standard models.

Semantic Web languages offer a flexible and standard method to represent several kinds of knowledge. For this reason the next section specifies an Architecture of Semantic Web Models. **However even in that case, these artifacts may depend on other unstructured or tacit knowledge.**

The Tables at the end of this section contain a few examples of different ways for representing (or just denoting) community knowledge.

However it should be remarked that the core model regarding intelligibility (modules and dependencies), i.e. like the one illustrated in Figure 2-10, is advantageous to be expressed in Semantic Web languages and will be detailed later on.

Finally we should say that CASPAR aims at providing insights to the problem of digital information preservation and to investigate the corresponding methodological issues (including the representation of DC knowledge). Its focus it not confined on formalizing the specific knowledge related to the specific datasets of the CASPAR testbeds.

Domain	**What could be a module**	**What could be a Profile**
Scientific Knowledge	Module = A scientific paper. Note that the knowledge expressed in the paper may is not possible to be formalized.	Profile = a set of modules. For instance it could be a set of 100 well known and foundamental papers of a discripline. This may mean that the gap of a new paper (i.e. that is needed to be recorded) could be a metadata record (annotation) that relates that paper with the aforementioned set of papers.
	Module = a dictionary. It	Profile = a set of modules. So a set of

	could be a dictionary comprising terms and textual descriptions of the terms. Examples: AAT, CIDOC CRM Ontology, FITS Dictionary. It could be expressed in the form of an RDF file (also containing the appropriate textual descriptions).	well known (consesual) dictionaries and ontologies.
	Module = an RDF KB (schemas + instantiations). It could have the form of one file in RDF/XML that contains both schema and data.	Profile = a set of RDF Kbs.
	Module = the knowledge of an individual person. We could consider DavidGiaretta as a module. We could have a dependency of the form: CASPAR > DavidGiaretta	Profile = a set of persons. For example the profile of a community could contain a set of key persons.
Contempora ry arts	Module = a MAX/MSP software	Profile = a set of certified tools for running MAX/MSP files
General case of Descriptive Metadata	Consider OAIS provencance. We may describe it by a sequence of derivations and transformations. In that case a module could be the schema that allows expressing these derivations (e.g. CIDOC CRM).	

Some examples of dependencies are shown next

Domain	Example Dependency Relations	
Software Engineering	The spec of an Information System according to MDA should depend on OMG startdards (instead of individual technologies). E.g. CASPARSpec > UML 2.0	
Scientific Background	We may say that a publication X depends on all of its citations Y. [Tzitzikas&Flouris, ECDL'2007] > [Tzitzikas, DEXA'2007]	
Multimedia Performance Data	AvisDeTempete.zip > Max/MSPsoftware. However that testbed is currently trying to formalize the logical structure of Max/MSP patches. In that way we will have AvisDeTempteNew.zip > StardardLogicalStructureOfMax/MSP. This will make it independent of the particular software.	
General	Suppose that an actor A1 wants to preserve the 2D layout of a web page. Another actor A2 may want to preserve also the behaviour of the page. How these differences are reflected to the dependencies? A1: He takes a screen dump and says mypageScreendump.jpg > JPG	

	A2: mypageDescriptionOfFlowOfControl > FLOWofControlStandard The latter (FLOWofControlStandard) could be an appropriate profile of UML.	
RDF	Note that each RDF file has a header that contain all its dependencies. So we may say that if t is an RDF file then its header lists all ements of C+(t).	
Java code	The signature of a java class contains its direct superclass (not the indirect). However the specialization hierarchy is not the only aspect of dependency in software code.	

More information about dependencies of INA's works are described in document

- "Representation Information Dependencies: Examples from INA" by Erik Gebers.

Some examples of gaps

Domain	What a Gap could be	
General case	A set of modules, i.e. the result of Gap(o,u)	
A more refined case (typed gaps)	The result of Gap(o,u,X) where X is a subset of the subtypes of Modules. For example we may want to get the missing modules that concern structural or algorithmic information.	
Ontology Diff	Consider the case where Tu contains CIDOCver1 and we have a new metadata record according to CIDOCver2. In that case the plain gap manager would return as a gap the module CIDOCver2. However we may employ in this case a more fine grained approach and assume that gap = diff(CIDOCver1-> CIDOCver2) or diff(CIDOCver1, CIDOCver2). The result of the former is a set of update operations that the actor could apply to his knowledge base in order to reach CIDOCver2. The result of the latter is a set of difference that the person responsible for the archive could investigate in order to judge whether he should migrate his archive CIDOCver2.	

4 ARCHITECTURE OF SEMANTIC WEB MODELS

4.1 THE BENEFITS OF ADOPTING SEMANTIC WEB TECHNOLOGIES

There are several options for implementing the above framework and the related services. 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 could be adopted in two places:

1/ for representing the core OAIS ontology and the information needed for implementing intelligibility-aware processes

2/ for representing the knowledge of a community in case this is available or possible.

4.2 CORE ONTOLOGY FOR OAIS AND INTELLIGIBILITY-RELATED TASKS

Figure 4-1 illustrates a possible architecture of SW schemas and data. The upper level comprises 3 small SW schemas. The basic dependency management services could need to know only these schemas. The bottom layer shows an indicative instantiation of the above schemas.

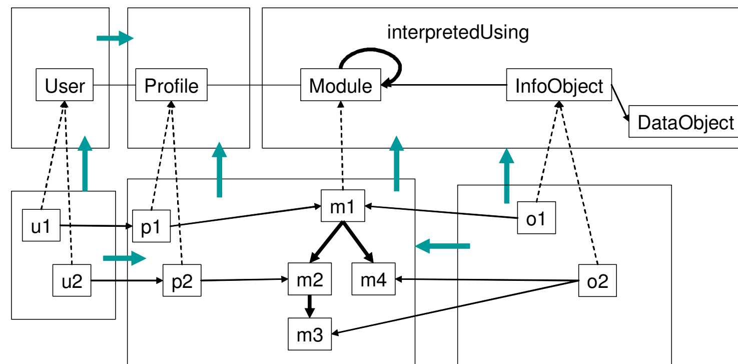


Figure 4-1 Architecture of Semantic Web Models

Between these two layers a number of other schemas can be defined that specialize/refine the elements of the upper schemas. For instance, the notion of module can be specialized (we could have a typology/taxonomy of modules). Furthermore the property class interpretedUsing can be specialized (the new specialized property class could have as domain and range subclasses of Module). This is illustrated in Figure 4-2.

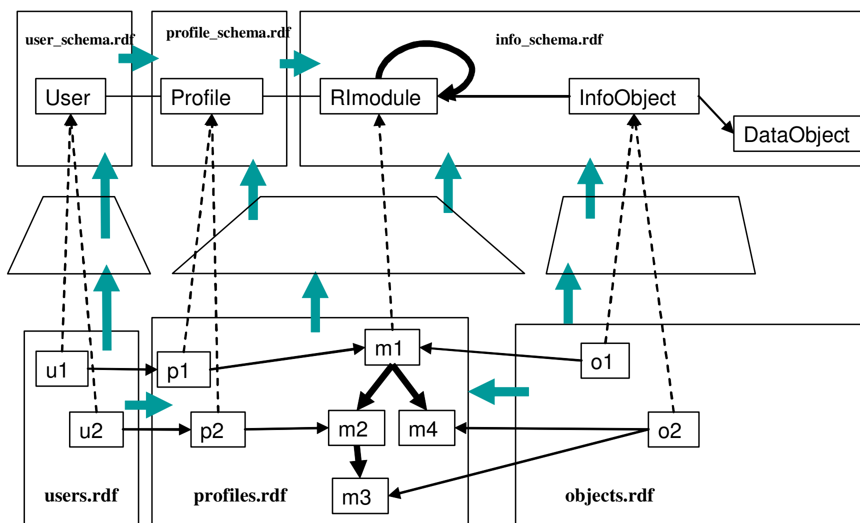
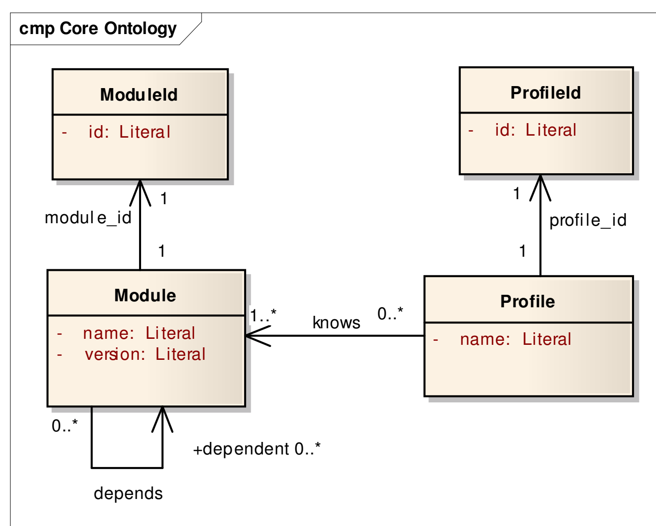


Figure 4-2 Extensible Modeling

One benefit of adopting Semantic Web technologies and an architecture of schemas like this, is that we can build a preservation system that needs to know only the upper schemas. To be more specific, this means that the queries may be formulated in terms of these schemas. However the same queries will function correctly even if the data level instantiates specializations of the above schemas. This is due to the semantics of specialization.

4.2.1 The Core Ontology for Exchanging Modules, Dependencies and DC Profiles

We have already defined (and expressed in RDFS) a Core Ontology for exchanging modules, dependencies and DC profiles. The detailed description of the ontology is given in a separate document. Its description as an UML class diagram is given in the next figure.



We have already used this ontology for describing various data.

PRONOM is an online registry of technical information. Specifically it provides information about the file formats, software products and other technical components required to support long-term access to electronic records and other digital objects of cultural, historical or business value. PRONOM holds information about file formats, and the software products which can process (read, write, identify etc) each format. Information related to the file formats, such as documentation about them, their compression types, character encoding schemes and intellectual property rights is also held. Currently 857 records are listed.

We have extracted these data and we have expressed them according to the core ontology.

4.3 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. 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. 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 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 Figure 4-3 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).

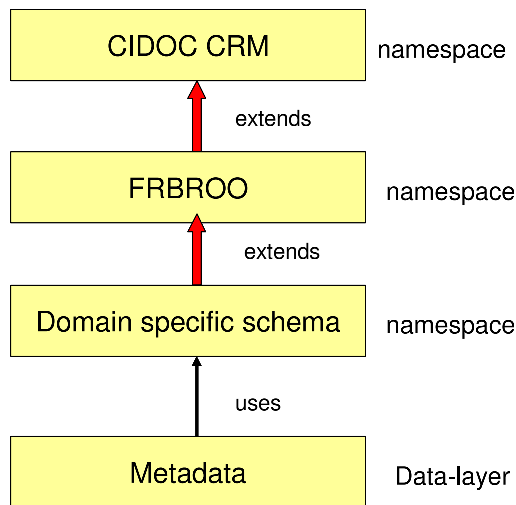


Figure 4-3 Architecture of Semantic Web Models

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 metata. Figure 4-4 illustrates the idea.

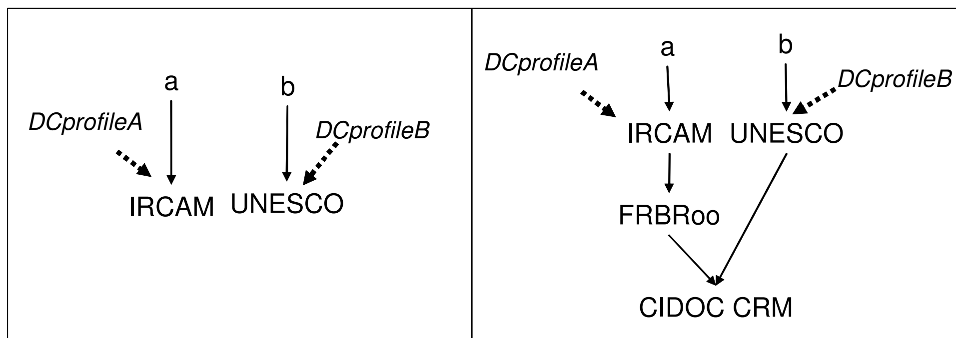


Figure 4-4 Cross-Community Interpretability

4.4 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.

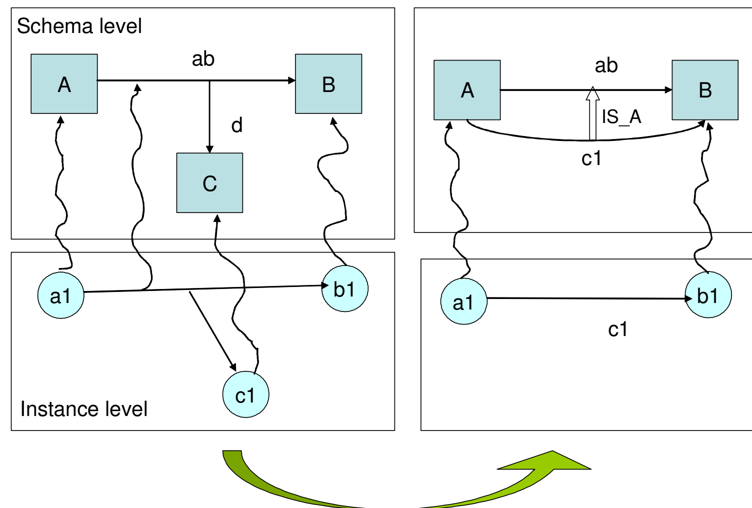


Figure 4-5 Links from Links

The left part of Figure 4-5 shows a conceptual diagram illustrating a part of an ontology plus an instantiation of it. In particular, there is a property **ab** having domain the class **A** and range the class **B**. There is a property **d** whose domain is the property **ab**, having range the class **C**. The figure shows also an instantiation of this schema, specifically **a1** is an instance of class **A**, **b1** is an instance of class **B**, and **c1** is an instance of class **C.**

The above logical structure should be implemented as the right side of Figure 4-5. Specifically, we add at our schema (preferable at the domain specific schema) another property named **ci** which is defined as subproperty (i.e. specialization) of the **ab** property.

The definition of **c1** can be placed at the domain specific schema as in Figure 4-6. Actually the right part of Figure 4-6 is a realization of the architecture described in Figure 4-3.

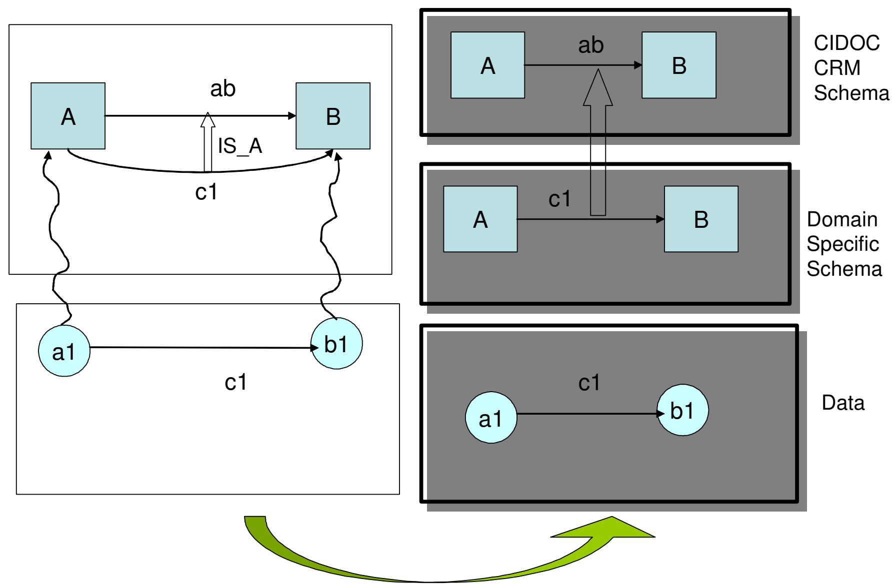


Figure 4-6 Partitioning the knowledge into 3 artifacts

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 4-7.

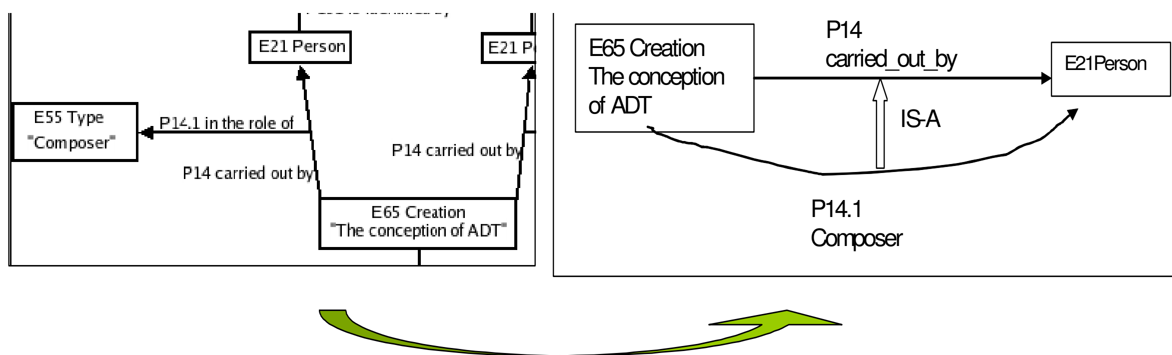


Figure 4-7 Example of Modeling Composer

4.5 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.

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.

4.6 KNOWLEDGE MANAGEMENT AND CASPAR TESTBEDS

Detailed information are given in "D4102 Integrated report of R&D activities on the Cultural, Performing Arts and Science data preservation testbeds". This section is supposed the summarize the key points.

4.7 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 modelling 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 farther" 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 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).

5 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 Manager (SWKM) could provide a set of core services for managing Semantic Web data. At the upper layer, a CASPAR Knowledge Manager (CKM) will provide high level services based on the OAIS model (and its extensions) aiming at offering an abstraction useful for preservation information systems. CKM can be 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 could be 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.

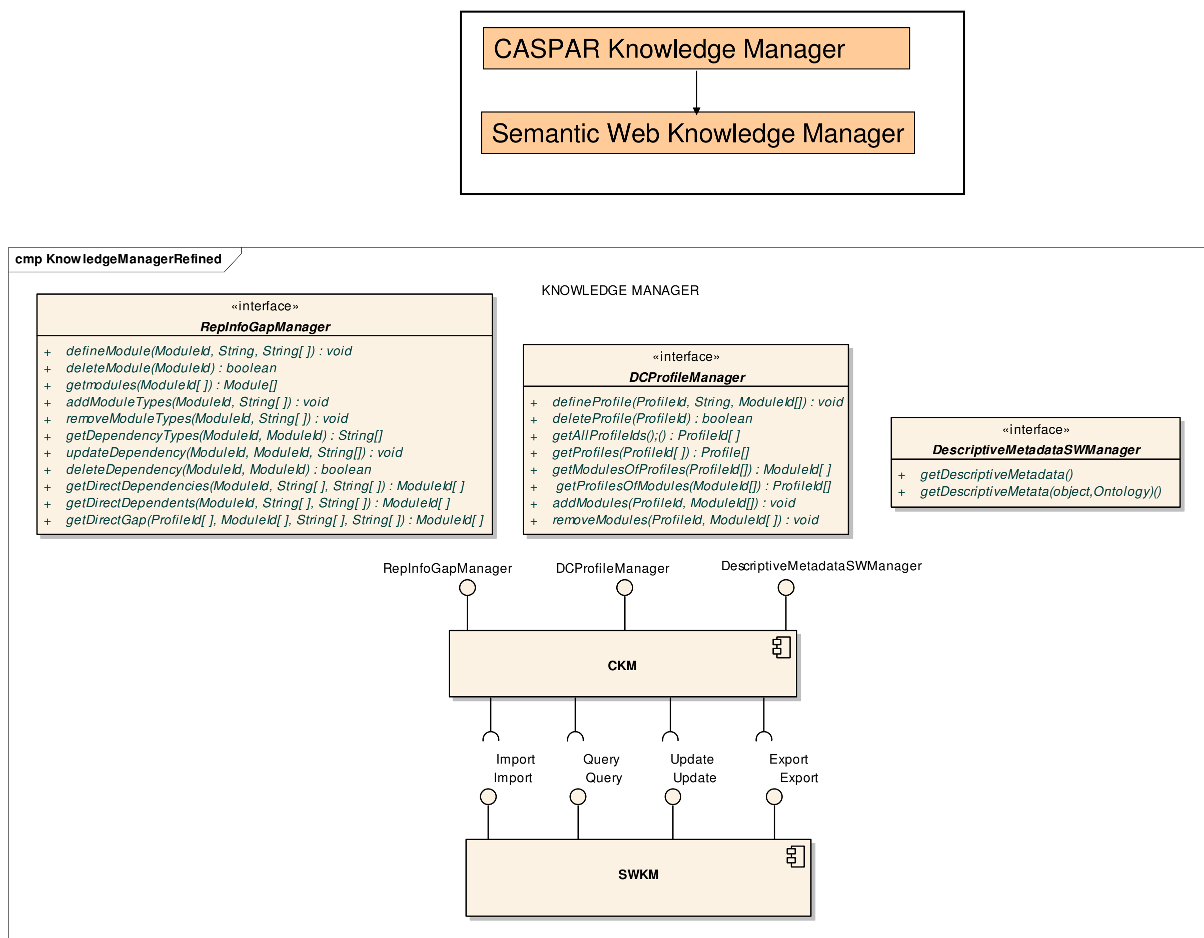


Figure 5-2 Component Model of Knowledge Manager

The description of the components follows

Component	Knowledge Manager
Responsibilities	Capture Higher level Semantics Manage Designated Community Knowledge profile Identify RepInfo 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** DCProfileManager RepInfoGapManager DescriptiveMetadataSWManager (B) **SWKM** Services: Query () Update () Import() Export() SWMainMemoryManagement SWMainMemoryModel
Required Interfaces	
Artefacts	The first version of the implementation of the (B)-Services, as a set of web services (based on RDFSuite), has been done. The implementation of a proof-of-concept (A)-level based on the (B)-layer is ongoing.

CKM aims at enabling the intelligibility-aware services that were described in Section 2. It will provide the above specified interfaces:

DCProfileManager

RepInfoGapManager

DescriptiveMetadataSWManager

The following tables sketch the operations of these interfaces. The detailed specification is given in a the subsequent sections.

Interface	DCProfileManager
Operations	defineDCProfile updateDCProfile deleteDCProfile getDCProfiles getDCProfileContent
Used by	Data Access Manager and Security

Interface	RepInfoGapManager
Operations	defineDependencies getDirectDependencies getAllDependencies updateDepenencies deleteDependencies getRequiredRepInfo(object) getMissingRepInfo(dcprofiles,objects)
Used by	Data Access Manager and Security Registry Preservation Orchestration Manager

Interface	DescriptiveMetadataSWManager
Operations	getDescriptiveMetadata(object) getDescriptiveMetadata(object, ontology)
Used by	Finding Aids

The low level (knowledge repository) basic services should provide scalable persistence services for large volumes of ontologies and ontology-based descriptions. Access and manipulation can be 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). The repository will be built upon the FORTH-ICS RDFSuite open source platform (available at http://139.91.183.30:9090/RDF/).

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.

Interface	SWKM Services
Operations	query update import export
Used by	Data Access Manager and Security Finding Aids

6 HIGH LEVEL SERVICES (ANALYTIC DESCRIPTION)

This section presents a more detailed description of the High Level Services. In particular it presents the interfaces of these services in a way that is independent of any particular implementation.

This specification is a more refined specification of what is presented in the Architecture deliverable.

Beans:

```
interface Profile {
    ProfileId getId();
    String getName();
    Module[ ] getKnownModules();
}
```

```
interface Module {
    ModuleId getId();
    String getName();
    String[] getTypes();
}
```

6.1 DC PROFILE MANAGER INTERFACE

```
// Defines a new profile and associates it with a set of modules (the modules assumed to be known)
// Throws ProfileIdAlreadyExistsException if the profile identifier already exists.
// Throws ModuleDoesNotExistException if any module identifier specified is non-existent.
// If the operation completes successfully, any other defineProfile operation which provides the
// same profileId will fail, unless deleteProfile(profileId) is called first.
void defineProfile(ProfileId profileId, String profileName, ModuleId[ ] knownModules)
        throws ProfileIdAlreadyExistsException, ModuleDoesNotExistException;
```

```
// Deletes a profile by its identifier, if it exists. Returns whether that profile is existent prior to this call.
boolean deleteProfile(ProfileId profileId);
```

```
// Returns the corresponding profiles of specified profile identifiers.
// Throws ProfileDoesNotExistException if a profile identifier is invalid.
Profile[ ] getProfiles(ProfileId[ ] profileIds)
        throws ProfileDoesNotExistException;
```

```
//Returns all existing profile identifiers (which have not been deleted).
ProfileId[ ] getAllProfileIds();
```

```
// Returns all known module identifiers (in no particular order) of a list of profiles.
//Throws ProfileDoesNotExistException if a profile identifier is invalid.
ModuleId[ ] getModulesOfProfiles(ProfileId[ ] profileIds)
    throws ProfileDoesNotExistException;
```

```
// Returns all known profile identifiers (in no particular order) which are currently associated with any module from the ones provided.
//Throws ModuleDoesNotExistException if a module identifier is invalid.
ProfileId[ ] getProfilesOfModules(ModuleId[ ] moduleIds)
    throws ModuleDoesNotExistException;
```

```
// Adds a list of modules identifiers in the set of known modules of a profile.
```

// Throws ProfileDoesNotExistException if the profile identifier is invalid.
// Throws ModuleDoesNotExistException if a module identifier is invalid.
void addModules(ProfileId profile, ModuleId[] modules)
throws ProfileDoesNotExistException, ModuleDoesNotExistException;

// Removes a list of modules identifiers from the set of known modules of a profile.
// Throws ProfileDoesNotExistException if the profile identifier is invalid.
// Throws ModuleDoesNotExistException if a module identifier is invalid.
void removeModules(ProfileId profile, ModuleId[ ] modules)
throws ProfileDoesNotExistException, ModuleDoesNotExistException;

6.2 REPINFO GAPMANAGER INTERFACE

// Creates a new module with a given identifier, name, and a set of types.
// Throws ModuleAlreadyExistsException if a module already exists with the same identifier.
void defineModule(ModuleId id, String name, String[ ] moduleTypes)
throws ModuleAlreadyExistsException;

//Returns the modules which correspond to a list of module identifiers.
//Throws ModuleDoesNotExistException if any module identifier does not exist.
Module[ ] getModules(ModuleId[ ] moduleIds)
throws ModuleDoesNotExistException;

// Deletes a module by its identifier. Returns whether the module identifier actually existed prior to this call. **boolean deleteModule(ModuleId module);**

// Adds the specified types to the types of a module.
// Throws ModuleDoesNotExistException if any module identifier does not exist.
void addModuleTypes(ModuleId module, String[ ] types)
throws ModuleDoesNotExistException;

// Removes the specified types from the types of a module.
// Throws ModuleDoesNotExistException if any module identifier does not exist.
void removeModuleTypes(ModuleId module, String[ ] types)
throws ModuleDoesNotExistException;

```
// Returns the types of the dependency relationship between moduleA and moduleB.
// Throws ModuleDoesNotExistException if either module does not exist, or
// DependencyDoesNotExistException if there is no direct dependency between these modules.
String[ ] getDependencyTypes(ModuleId moduleA, ModuleId moduleB)
        throws DependencyDoesNotExistException, ModuleDoesNotExistException;
```

```
// Updates or creates a new dependency (if there was none between moduleA and moduleB)
// between moduleA and moduleB.
//The dependency's types after the successful invocation of this method will be the ones specified.
//Throws ModuleDoesNotExistException if either module does not exist.
void updateDependency(ModuleId moduleA, ModuleId moduleB, String[] types)
  throws ModuleDoesNotExistException;
```

```
// Deletes a dependency of moduleA to moduleB. Returns whether such a dependency existed
// prior to this call.
      boolean deleteDependency(ModuleId moduleA, ModuleId moduleB);
```

```
// Returns a subset of the modules that a specified module directly depends upon. A module X is
// included in the returned set if and only if all of the following conditions hold:
//  (a) a dependency of the form (module depends on X) has been defined and still exists
//  (b) X contains at least one type included in acceptableModuleTypes,
//      or acceptableModuleTypes is empty
//      (c) the types of the dependency relationship (module, X) contain at least one element of acceptableDependencyTypes,
//       or acceptableDependencyTypes is empty.
//  Throws ModuleDoesNotExistException if a non-existent module is specified.
ModuleId[ ] getDirectDependencies(ModuleId module, String[ ] acceptableModuleTypes,
String[ ] acceptableDependencyTypes)
  throws ModuleDoesNotExistException;
```

```
// Returns a subset of the modules which directly depend on the specified module.
// A module X is included in the returned set if and only if:
//  (a) a dependency of the form (X depends on module) has been defined and still exists
//  (b) X contains at least one type included in acceptableModuleTypes, or acceptableModuleTypes is empty,
//  (c) the types of the dependency relationship (module, X) contains at least one element of acceptableDependencyTypes, or acceptableDependencyTypes is empty.
//Throws ModuleDoesNotExistException if a non-existent module is specified.
ModuleId[ ] getDirectDependents(ModuleId module, String[ ] moduleTypes, String[ ] dependencyTypes)
  throws ModuleDoesNotExistException;
```

```
//  Returns the direct gap of modules which are needed by a set of profiles to understand a set of modules,
//  filtered by module and dependency types. This is equivalent to:
//  - Calculating the union F of getDirectDependencies(m, moduleTypes, dependencyTypes), of
//     all m in modules.
//   - Returning F minus getModulesOfProfiles(profileIds)
//  Throws ProfileDoesNotExistException or ModuleDoesNotExistException if a non-existent profile or
//   module is specified, respectively.
ModuleId[ ] getDirectGap(ProfileId[ ] profileIds, ModuleId[ ] modules, String[ ] moduleTypes, String[ ] dependencyTypes )
     throws ProfileDoesNotExistException, ModuleDoesNotExistException;
```

Note that currently we under-specify the semantics of [module, dependency] types (currently modelled as strings). We expect that in the future we will understand more thoroughly the level of detail that would be most profitable for defining and manipulating types.

Changes in Dependency Graphs

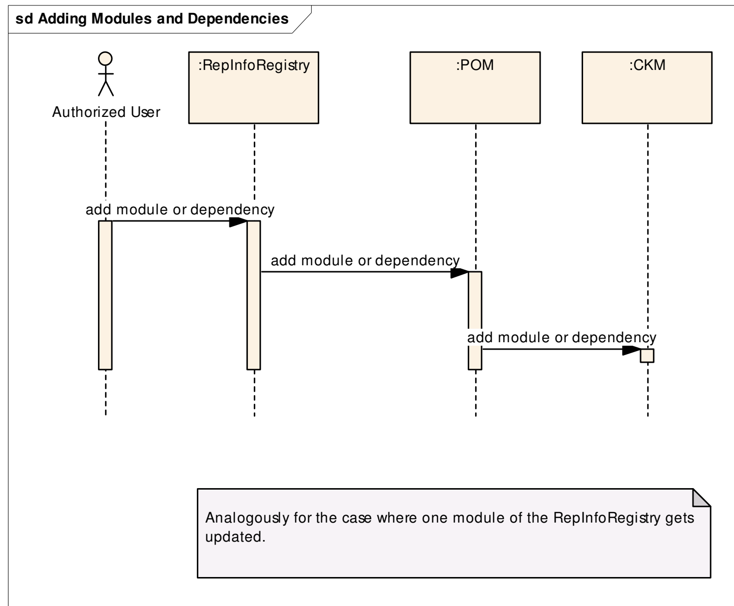
The interfaces **DCProfileManager** and **RepInfoGapManager** are going to be used by the **POM** (Preservation Orchestration Manager). One aspect that has to be analyzed in more detail is the impact of changes on the modules/dependencies/profiles. POM is responsible for the corresponding notification services but what services of CKM are needed has to be analyzed in more detail. Some results on this issue, are available at [S2] (Y. Tzitzikas, G. Flouris: Mind the (Intelligibility) Gap. 11th European Conference on Research and Advanced Technology for Digital Libraries, ECDL'2007, Budapest, Hungary, September 2007). However a more detailed technical specification is required. For instance we have to investigate if the warnOfImpliedChange (that POM is requested from CKM)

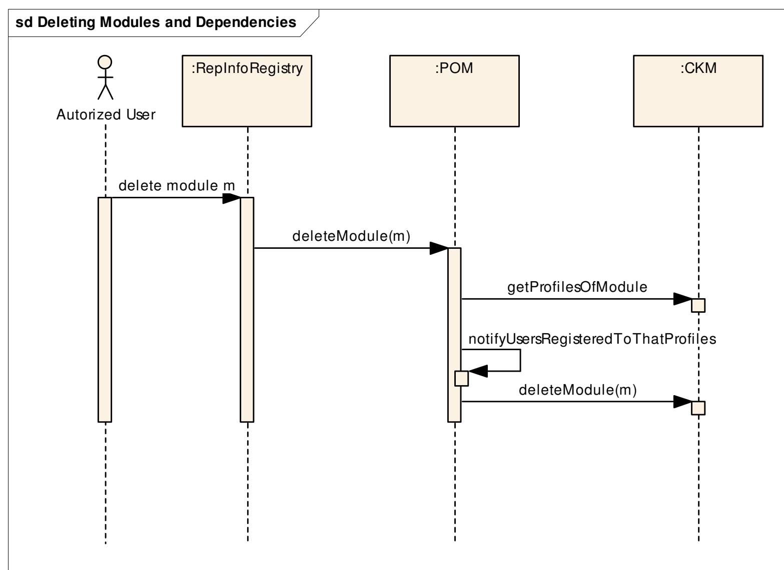
can be based on getDirectDependents(ModuleId module, String[ ] moduleTypes, String[ ] dependencyTypes) or some additional calls. A preliminary description (on the basis of [S2]) follows:

Simple Operations	Invariants	General Comments
Deletion of module t	What happens to the graph? All deps to and from t should be deleted? See [S2]	If we assume that the call deleteModule(t) is issued by POM, then POM may also have to call some other methods of RepInfoGapManager so that to identify the affected DC profiles.
Deleting dependencies from the model		Probably there is no side-effect
Adding dependencies to the model		.
Complex Operations		
Upgrade a module with a backwards compatible module It preserves the deps of the previous version	See [S2]	

Another issue is how the **RepInfoGapManager** gets notified when something changes (e.g. additions of modules and dependency relationships). **RepInfoRegistry** and **POM** are the related components. One reasonable approach is the following: Whenever a new module in added at the **RepInfoRegistry** (or any other change), the RepInfoRegistry informs POM and then **POM** informs **RepInfoGapManager**. Then the POM could issue the appropriate calls to CMK in order to identify who other actors should be notified.

Indicative interaction diagrams follow.


sd Adding Modules and Dependencies
Authorized User
:RepInfoRegistry
:POM
:CKM
add module or dependency
add module or dependency
add module or dependency
Analogously for the case where one module of the RepInfoRegistry gets updated.


sd Deleting Modules and Dependencies
Autorized User
:RepInfoRegistry
:POM
:CKM
delete module m
deleteModule(m)
getProfilesOfModule
notifyUsersRegisteredToThatProfiles
deleteModule(m)

6.3 DESCRIPTIVE METADATA SW MANAGER INTERFACE

In general the Descriptive Metadata Services are the Basic SW KM Services that are described in Section 7.

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

Service	Requested By	
getRelatedConcepts(Concept, RelationshipType): Concepts[]	**Virtualization**	
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:

Methods	**Requested From**	**Comments**
getDescriptiveMetadata(object)	**FindingAids**	
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.

6.3.1 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.

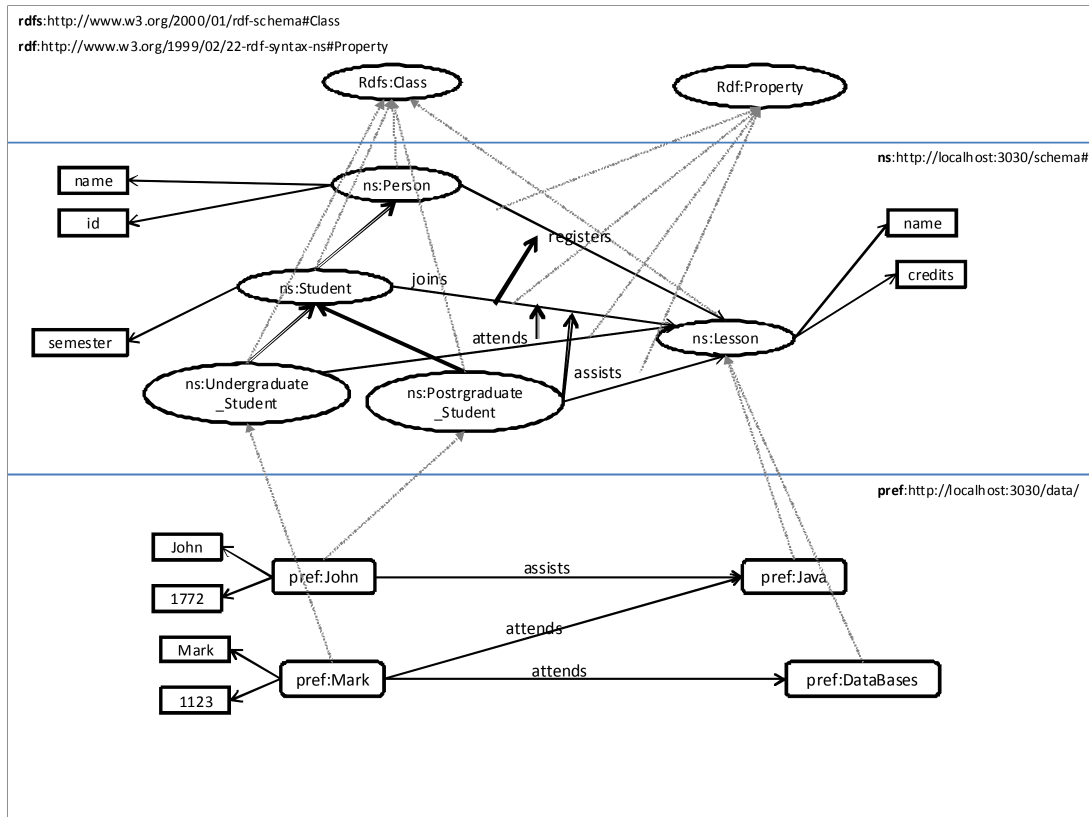
ServiceType	conceptType	Result
CLASSES	CLASS	The metaclass URIs under which the given class is classified
	PROPERTY_CLASS	The metaclass URI under which the given property is classified
	CLASS_INSTANCE	The class URIs under which the specified instance is classified
INSTANCES	CLASS	All Instances of the given Class URI
	PROPERTY_CLASS	All Instances of the given Property URI
	CLASS_INSTANCE	nothing
ALLSUB	CLASS	All subClass URIs of the given class URI
	PROPERTY_CLASS	All subProperty URIs of the given property URI
	CLASS_INSTANCE	nothing
DIRSUB	CLASS	The direct subClass URIs of the given class URI

	PROPERTY_CLASS	The direct subProperty URIs of the given property URI
	CLASS_INSTANCE	nothing
ALLSUP	CLASS	All superClass URIs of the given property URI
	PROPERTY_CLASS	All superProperty URIs of the given property URI
	CLASS_INSTANCE	nothing
DIRSUP	CLASS	The direct superClass URIs of the given property URI
	PROPERTY_CLASS	The direct superProperty URIs of the given property URI
	CLASS_INSTANCE	nothing
FROM	CLASS	All property URIs that have the specified Class as domain
	PROPERTY_CLASS	The domain of the specified Property
	CLASS_INSTANCE	nothing
TO	CLASS	All property URIs that have the specified class as range
	PROPERTY_CLASS	The range of the specified Property
	CLASS_INSTANCE	nothing
PATTERN_MATCH	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

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.

Short Example



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*)**
    - ["http://localhost:3030/data/Mark" , "http://localhost:3030/data/Java"]
    - ["http://localhost:3030/data/Mark" , "http://localhost:3030/data/Databases"]
- **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"

  - "http://localhost:3030/schema#Postgraduate_Student"
  - "http://localhost:3030/schema#Undergraduate_Student"
- **get("http://localhost:3030/schema#registers",ServiceType.*ALLSUB*,ConceptType.*PROPERTY_CLASS*)**
  - "http://localhost:3030/schema#joins"
  - "http://localhost:3030/schema#attends"
  - "http://localhost:3030/schema#assists"
- **get("http://localhost:3030/data/Java",ServiceType.*ALLSUB*,ConceptType.*CLASS_INSTANCE*)**
  - [ ]
- **get("http://localhost:3030/schema#Person",ServiceType.*DIRSUB*,ConceptType.*CLASS*)**
  - "http://localhost:3030/schema#Student"
- **get("http://localhost:3030/schema#registers",ServiceType.*DIRSUB*,ConceptType.*PROPERTY_CLASS*)**
  - "http://localhost:3030/schema#joins"
- **get("http://localhost:3030/data/Mark",ServiceType.*DIRSUB*,ConceptType.*CLASS_INSTANCE*)**
  - [ ]
- **get("http://localhost:3030/schema#Undergraduate_Student",ServiceType.*ALLSUP*,ConceptType.*CLASS*)**
  - "http://localhost:3030/schema#Student"
  - "http://localhost:3030/schema#Person"
- **get("http://localhost:3030/schema#assists",ServiceType.*ALLSUP*,ConceptType.*PROPERTY_CLASS*)**
  - "http://localhost:3030/schema#joins"
  - "http://localhost:3030/schema#registers"
- **get("http://localhost:3030/data/Mark",ServiceType.*ALLSUP*,ConceptType.*CLASS_INSTANCE*)**
  - [ ]
- **get("http://localhost:3030/schema#Undergraduate_Student",ServiceType.*DIRSUP*,ConceptType.*CLASS*)**
  - "http://localhost:3030/schema#Student"
- **get("http://localhost:3030/schema#assists",ServiceType.*DIRSUP*,ConceptType.*PROPERTY_CLASS*)**
  - "http://localhost:3030/schema#joins"
- **get("http://localhost:3030/data/Mark",ServiceType.*DIRSUP*,ConceptType.*CLASS_INSTANCE*)**
  - [ ]
- **get("http://localhost:3030/schema# Student",ServiceType.*FROM*,ConceptType.*CLASS*)**
  - "http://localhost:3030/schema#registers"
  - "http://localhost:3030/schema#joins"
- **get("http://localhost:3030/schema#assists",ServiceType.*FROM*,ConceptType.*PROPERTY_CLASS*)**
  - "http://localhost:3030/schema#Postgraduate_Student"
- **get("http://localhost:3030/data/Java",ServiceType.*FROM*,ConceptType.*CLASS_INSTANCE*)**
  - [ ]
- **get("http://localhost:3030/schema# Student",ServiceType.*TO*,ConceptType.*CLASS*)**
  - [ ]
- **get("http://localhost:3030/schema#assists",ServiceType.*TO*,ConceptType.*PROPERTY_CLASS*)**
  - "http://localhost:3030/schema#Lesson"
- **get("http://localhost:3030/data/Java",ServiceType.*TO*,ConceptType.*CLASS_INSTANCE*)**

    - [ ]
- **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("tt",ServiceType.*PATTERN_MATCH*,ConceptType.*CLASS_PROPERTY*)**
    - "http://localhost:3030/schema#a**tt**ends"
- **get("a",ServiceType.*PATTERN_MATCH*,ConceptType.*CLASS_INSTANCE*)**
    - "http://localhost:3030/data/M**a**rk"
    - "http://localhost:3030/data/J**a**va"
    - "http://localhost:3030/data/D**a**tabases"

7 BASIC SWKM SERVICES (ANALYTIC DESCRIPTION)

7.1 PURPOSE AND SCOPE

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

7.2 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 obejct. 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 (disjoiness 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.

7.3 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 are currently implementing 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) to provide a simple communication channel with CASPAR applications not featuring sophisticated SW update and query functionality (see import/export services).

7.4 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.

7.5 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 intelligility-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.

Import Services	
void store(**String[] URIs, String[] documents, String format)**	
void storeWithDependencies(**String[] URIs, String[] documents, String format)**	
Export Services	
String[][] fetch(**String[] URIs, String format)**	
String[][] fetchWithDependencies(**String[] uris, String format)**	
Query Services	
String query(**String queryString, String format)**	
Update Services	
boolean update(**String rulQuery**)	

7.6 IMPORTER SERVICE

7.6.1 Store

Signature:

```
void store(String[] URIs, String[] documents, String format)
```

Description:

Imports in the underlying storage the specified namespaces and/or graphspaces.

Each element in URIs array corresponds to an element in documents array, where as a document we define the corresponding namespace or graphspace. The format parameter must have the value of either "TRIG" or "RDF_XML". All elements of documents array are strings in the format specified by the format parameter. Henceforth, the corresponding URIs element of an element in documents will be referred to as "the URI of the document".

If a document defines an RDF namespace, the URI of that namespace will be the URI of the document. In case of TRIG format, the document may define one or more RDF graphspaces, which are locally identified by a graph id. Each graphspace's URI will be created by concatenating the document's URI and the graph id. These resulting URIs can be used to retrieve back the stored RDF spaces.

In case of TRIG format, at most one namespace is allowed per document (that namespace's URI will be the respective URI specified for the document), but there is no limitation in the number of graphspaces. In case of RDF_XML format, each document can have at most one namespace and at most one graphspace (possibly both). The namespace will get the URI of the document. The graphspace will be unnamed, i.e. there will be no way to fetch it through the Exporter service, but it is still accessible through the Query service. If the intention is to store a graphspace, TRIG format must be used. The two file formats and their capabilities are depicted in Figures 3 and 4 in Appendix A.

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).

If the underlying storage already contains a namespace or graphspace identified by a URI that this operation was about to create, the pre-existing RDF space is kept and used and cannot be superseded by any RDF space with the same URI.

Each RDF space may *depend* on other RDF spaces. Section 2.1 explains the notion of dependencies among RDF spaces. This operation resolves dependencies *passively*.

Validation of all resolved namespace and/or graphspaces will always take place before actual importing. If validation fails, an error will be raised, and the operation will be cancelled. It is also noted that the operation semantics are all-or-nothing, i.e. in case of a failure (for example, due to corrupted data, or a validation failure), no partial storing will take place; either all provided namespaces and graphspaces will be stored in the underlying storage, or none at all.

7.6.2 Store with Dependencies

Signature:

```
void storeWithDependencies(String[] URIs,
                           String[] documents,
                           String format)
```

Description:

Imports in the underlying storage the specified namespaces and/or graphspaces, along with their dependencies. Section 2.1 explains the notion of dependencies among RDF spaces. This operation resolves dependencies *eagerly*.

Each element in URIs array corresponds to an element in documents array. The format parameter may have the value of either “TRIG” or “RDF_XML”. All elements of documents array are strings in the format specified by the format parameter. Henceforth, the corresponding URIs element of an element in documents will be referred to as “the URI of the document”.

If a document defines an RDF namespace, the URI of that namespace will be the URI of the document. In case of TRIG format, the document may define one or more RDF graphspaces, which are locally identified by a graph id. Each graphspace’s URI will be created by concatenating the document’s URI and the graph id. These resulting URIs can be used to retrieve back the stored RDF spaces.

In case of TRIG format, at most one namespace is allowed per document (that namespace’s URI will be the respective URI specified for the document), but there is no limitation in the number of graphspaces. In case of RDF_XML format, each document can have at most one namespace and at most one graphspace (possibly both). The namespace will get the URI of the document. The graphspace will be unnamed, i.e. there will be no way to fetch it through the Exporter service, but it is still accessible through the Query service. If the intention is to store a graph, TRIG format must be used. The two file formats and their capabilities are depicted in Figures 3 and 4 in Appendix A.

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*.

If the underlying storage already contains a namespace or graphspace identified by a URI that this operation was about to create, the pre-existing and the to-be-stored RDF space are compared. If there are dimmed incompatible (for example, a domain of a property has changed), an error is raised, and the whole operation is cancelled. If no error occurs, the extra features of the new RDF space are added to the pre-existing, but features missing from the new are not removed from underlying storage.

Each RDF space may *depend* on other RDF spaces. Section 2.1 explains the notion of dependencies among RDF spaces. This operation resolves dependencies *eagerly*.

Validation of all resolved namespace and/or graphspaces will always take place before actual importing. If validation fails, an error will be raised, and the operation will be cancelled. It is noted that the operation semantics are all-or-nothing, i.e.in case of a failure (for example, due to corrupted data, or a validation failure), no partial storing will take place; either all provided namespaces and graphspaces will be stored in the underlying storage, or none at all.

7.7 EXPORTER SERVICE

7.7.1 Fetch

Signature:

```
String[][] fetch(String[] uris, String format)
```

Description:

Fetches the RDF contents of the requested URIs from the underlying storage, using the requested format. Accepted formats are: “TRIG”, “RDF_XML”.

Preconditions:

- Format is either “TRIG” or “RDF_XML”

Effects:

Only the requested namespaces and/or graphspaces are returned; not their dependencies. Section 2.1 explains the notion of dependencies among RDF spaces.

An error is raised if a requested URI cannot be located (i.e. it has not been stored in the past). 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.

7.7.2 Fetch with Dependencies

Signature:

```
String[][] fetchWithDependencies(String[] uris, String format)
```

Preconditions:

- Format is either “TRIG” or “RDF_XML”

Description:

Fetches the RDF contents of the requested URIs from the underlying storage *and all their dependencies*, using the requested format. Accepted formats are: “TRIG”, “RDF_XML”.

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). Section 2.1 explains the notion of dependencies among RDF spaces. 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.

7.7.3 Fetch with data

Signature:

```
String[][] fetchWithDependencies(String[] uris, String format)
```

Preconditions:

- Format is either “TRIG” or “RDF_XML”

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). The URI of these extra documents are created by appending “_data” to the URI of the namespace. For example, if the URI of the namespace is:

http://example.org/some/namespace#

The created URI of the related data document will be:

http://example.org/some/namespace_data#

Accepted formats are: “TRIG”, “RDF_XML”.

Effects:

Only the requested namespaces and/or graphspaces are returned; not their dependencies. Section 2.1 explains the notion of dependencies among RDF spaces. An error is raised if a requested URI 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 a returned document, and the document itself.

7.7.4 Fetch with data and dependencies

Signature:

```
String[][] fetchWithDataandDependencies(String[] uris, String format)
```

Preconditions:

- Format is either "TRIG" or "RDF_XML"

Description:

Fetches the RDF contents of the requested URIs from the underlying storage, as "Fetch with dependencies" 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). The URI of these extra documents are created by appending "_data" to the URI of the namespace. For example, if the URI of the namespace is:

http://example.org/some/namespace#

The created URI of the related data document will be:

http://example.org/some/namespace_data#

Accepted formats are: "TRIG", "RDF_XML".

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 a returned document, and the document itself.

7.8 QUERY SERVICE

Signature:

```
String query(String queryString, String format)
String[] query(String[] queryStrings[], String format)
```

Description:

Evaluates an RQL query, and returns the results using the requested format. Accepted formats are: "TRIG", "RDF_XML".

In the case of the multiple queries signature, the supplied queries run in parallel but they should all finish before the results are returned to the client. The operation is not atomic and thus if a query fails for any reason the rest will be executed normally and the array of results will return an array of results in the specified format, including the one(s) resulting from an error (where the error code will be provided).

Preconditions:

- Format is either “TRIG” or “RDF_XML”

Effects:

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

Demo:

A demo/tutorial of RQL is available at: http://139.91.183.30:3026/RQLdemo/

7.9 UPDATE SERVICE

Signature:

```
boolean update(String rulQuery)
boolean[] update(String[] rulQueries)
```

Description:

Evaluates an RUL query, and returns true upon successful execution, false otherwise.

In the case of the multiple updates signature, the supplied updates run in parallel, so the order of the actual execution/completion at the repository is not guaranteed but they should all finish before the results are returned to the client. Thus the set of the queries is not executed as an atomic operation. The client receives an array of true or false as a result corresponding to the successful or not completion of each individual query. If the user wants to guarantee the order of execution of the supplied queries, (s)he should supply them one by one using the single query interface or all of them in one combined RUL update statement (the latter is also handled as a single transaction by the repository).

Preconditions:

- None

Effects:

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

Demo:

A demo/tutorial of RUL is available at: http://139.91.183.30:3026/RULdemo/demo//

7.10 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.

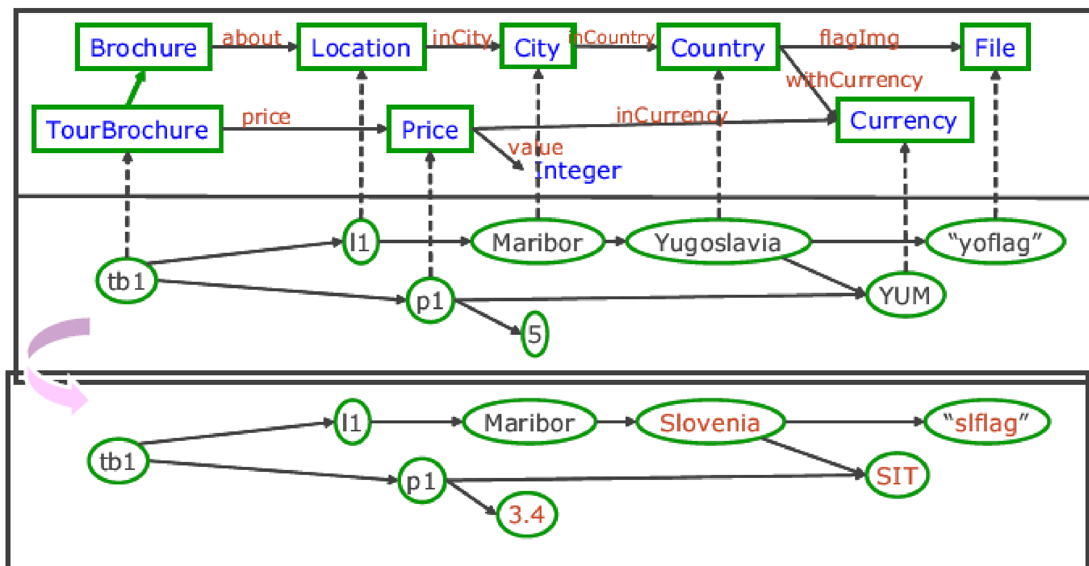


Figure 7-1 Example of Metadata Evolution

To update the data layer of the knowledge base (as shown at the bottom part of Figure 7-1) 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.

8 CASPAR KNOWLEDGE MANAGER COMPONENT

8.1 ITERATIONS

The main functionality of each iteration and some testing/validation methods are described in the following table.

Iteration	Functionality	Motivation	Testing/Validation Method(s)
I1	First version of the Basic SWKM Services. Options: 1/ Full installation 2/ Installation of only the client.	The high level knowledge management services (e.g. RepInfoGap Manager) will be based on the these. In addition, the rest CASPAR components may exploit these basic services.	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.
I2	(2a) Revised version of the Basic SWKM Services including a design and a first implementation of a Main Memory Model. (2b) First version of the Gap Manager	RepInfo Gap Manager is instrumental for providing intelligibility-aware services	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 this deliverable regarding intelligibility gaps. Definition of modules, profiles, dependencies, etc. Demonstrating examples that users with different profiles get different responses.
I3	Second version of Gap Manager	Interaction with other caspar components. GapManager should actually receive requests from other components. It is not expected to call other components.	I3TC1/ The registry or POM feeds the KM Repository (with modules and dependencies). I3TC2/ The component Preservation Orchestration Manager uses the RepInfoGapManager services.
I4	Diff over Knowledge Bases	Support of knowledge evolution is important and it should be demonstrated.	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.

Table 1 Functionality of each iteration

8.2 MORE DETAILS ON TESTING

Each test scenario will be checked by writing the corresponding JUnit tests. In this way it will be 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.

I1TC1	**No**	**Ability to build a SW repository offering persistence, validation and query and update services.**	
		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.	
		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.	
		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.	
		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.	
		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.	
		Try all combinations of pre/post conditions of the specification of the basic services (that are described in Section 7)	
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).**	
		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 in 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.	
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.**	
		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.	
I2aTC1		**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.	
I2aTC2		**Provision of an 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.	
I2bTC2		**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 6 a test case will be created on the basis of the pre/post-conditions that are already specified.	
I3TC1		**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: I2bTC2). The messages received from POM should be manageable by RepInfoGapManager.	
		POM is notified by the RepInfoRegistry 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.	
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.**	
		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

Timescale

Iteration	Input	Start Date	End Date
I1	All CASPAR deliverables	M6	M23
I2	D2101B	M18	M27
I3	Revised architecture	M28	M31
I4	Instantiations of CIDOC CRM and its extensions.	M28	M35

Table 3 Schedule of Iterations

Other useful remarks/actions:

Testbeds: To provide appropriate front-ends (GUIs) customized to their needs (to feed, retrieve and update data stored in the SW Repository).

Finding Aids:: to provide a general purpose graphical query interface based on ontologies

RepInfoRegistry: To be able to export its contents in RDF/XML or TRIG format so that to be possible to be imported at the KM repository. In addition it should be able to communicate with POM so that POM issues the appropriate calls to RepInfoGapManager. The latter is approach is more

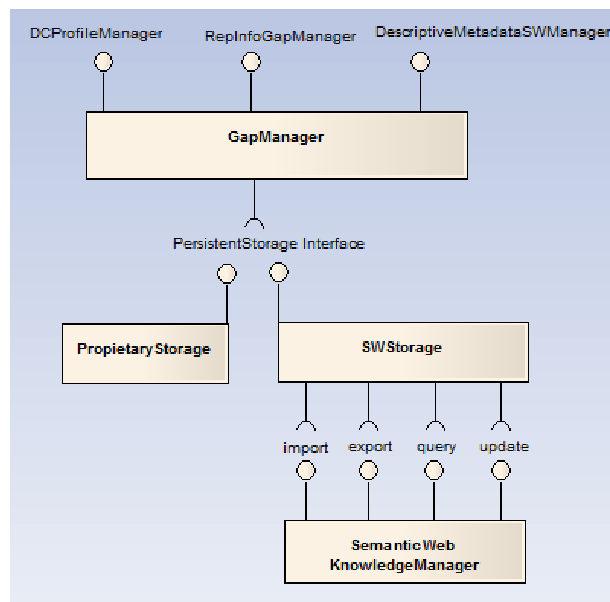
appropriate as it is reasonable to expect POM as the "mediator" component for exchanging messages between the various CASPAR components.

More detailed information will be available at http://wiki.casparpreserves.eu/bin/view/Main/2103GapManager

8.3 IMPLEMENTATION DETAILS

Gap Manager

The implementation has a modular design that enables changing the persistence layer easily. The current implementation has two persistence storage managers: one plain file-system based and another one over the Semantic Web Knowledge Middleware (over the basic SWKM services). The design of this module is illustrated below.



The current implementation supports more than one input and output formats.

- Proprietary Format

This is a simple, readable format. Three different files are used (for modules, profiles and dependencies) without having any information about the structure. The only information kept at these files concerns the real data. At the modules file each line corresponds to one module. At this line we keep all the information needed using simple syntax and separate different attributes of a module using tabs and write/read them in a specific order. Specifically the first attribute is the module identifier. After that is the name and the version of the module. The rest of the line is to keep the types a module holds. All the above are separated using tabs ('\t'). The dependencies file contains all the dependencies between modules described previously. The syntax here is the same as before. The two first attributes are the involving modules (using their identifiers) in the relation and the rest of it is to keep

information about the evolving dependency types (can be more than one). At profiles files, similarly to modules, each line represents one profile. The profile identifier and name are the first in the line and until the end of it we keep all the modules (using their identifiers) that the particular profile knows.

- Pronom Format

Using the xml file exported from Pronom we defined a utility to import data from this file format. The data we can get concern file formats and software. Specifically there are two different xml files (coming from Pronom). We use our utility to import data from these files and get them as modules, since we recognised both file format and software as modules. There are information that we do not currently use (such as external signatures, risks, and internal identifiers for file formats and software) at these files.

- Ontology for Modules, Dependencies and DC Profiles

We defined an ontology (in RDFS) for describing the modules, the dependencies between them and the DC profiles. This ontology consists of two schema files and other files containing instances. The first file (module_schema.rdfs) contains information about the schema of the modules and their dependencies. The other file (profile_schema.rdfs) describes the schema of the DC profiles. Any other file contains instances according to these schema files. More information can be found at S1.

The class diagram is shown in the next Figure.

class casparpreserves

8.4 CASPAR KM INSTALLATION

8.4.1 Required software components

The Prototype (V0.9) 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.

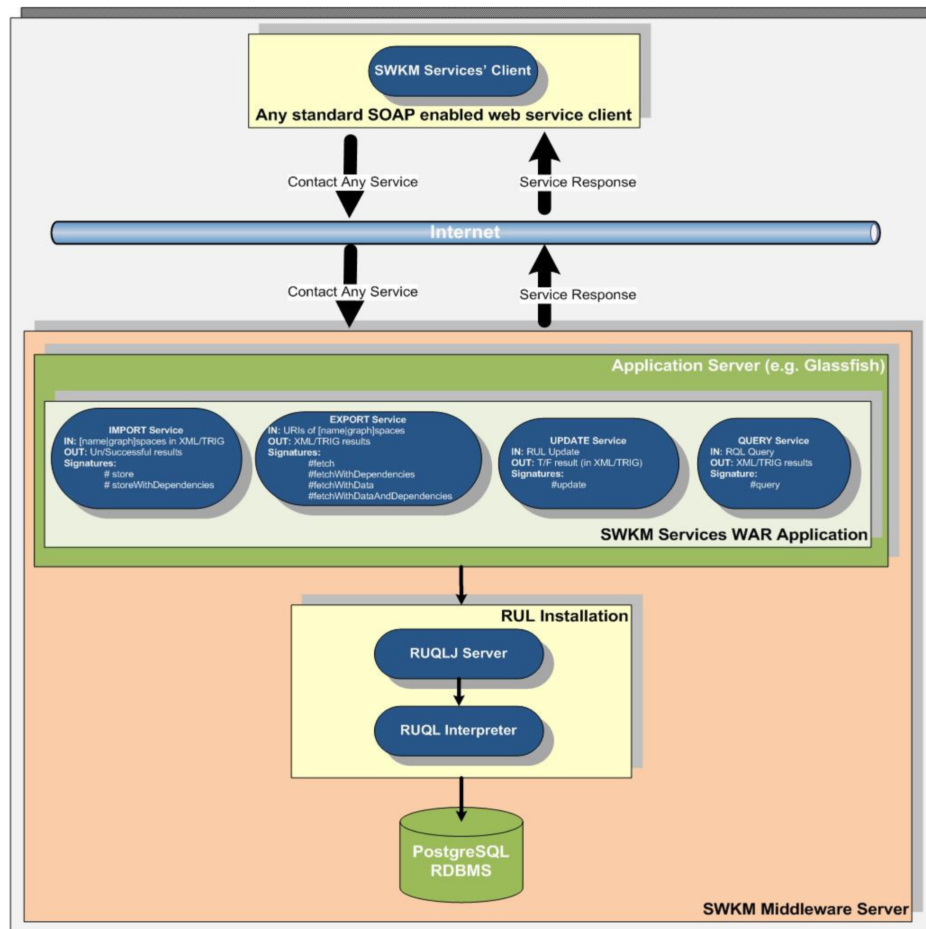


Figure 1. **The various software components required by the SWKM services**

In detail the required components for SWKM services to run are (their various connections and dependencies as well as the available services after deployment are depicted in Figure 1):

- A working installation of the PostgreSQL database server
  - Version 8.0 or higher
  - It can be downloaded from: http://www.postgresql.org/
  - Installation instructions at: http://139.91.183.30:9090/RDF/RSSDB/postgresql7_3_3_install.html
- The current release of RUL (RDF Update Language) including:
    - The RUL interpreter (included in RUL release) and
    - The RUQLJ server (included in RUL release)
  - Currently at version 1.0
  - It can be downloaded from: http://139.91.183.30:9090/RDF/RUL/Register.html
  - Installation instructions at: http://139.91.183.30:9090/RDF/RUL/Install.html

A servlet container or an application server eg:

- Sun Java System Application Server Platform Edition 9.0_01 (build b02-p01) – shortly called "Glassfish"
- Available for download from: https://glassfish.dev.java.net/downloads/v1_ur1-p01-b02.html .Installations instructions can be found at the same page.

- The SWKM web application (Java) deployed in the servlet container (or the application server).

8.4.2 The SWKM services as a web application

The services are packaged in the form of a single WAR (Web Application aRchive) file, which can be downloaded from

http://wiki.casparpreserves.eu/bin/view/Main/2103GapManager

(filename:*CASPAR_SWKM_WS.war*)

and is directly deployable to any servlet container or application server (such as Tomcat or Glassfish; this file also is ZIP-compatible so it can be opened by any ZIP-capable utility if its contents need to be inspected or changed).

However the most latest releases of software and documentation for the Semantic Web middleware (basic services and client) will be continuously available at http://athena.ics.forth.gr:9090/SWKM/.

Each servlet container may have a different process for deploying, though. To deploy the web services application in "Glassfish" access the "Admin Console" at:

http://your_App_Server_IP:4848/

then in the "Application" section, select "Web Applications", press the "Deploy" button, browse to the local folder where the .war file is placed and select it.

In Tomcat, just place the file inside the "webapps" folder of Tomcat installation folder. A directory will be automatically created where the services will be automatically deployed.

If a need to reconfigure the web application occurs then a file named:

WEB-INF/classes/applicationContext.xml

exists which contains the necessary configuration parameters and which is a Spring (a full-stack Java/JEE application framework used here mostly to support assembling and configuration of components, http://www.springframework.org/) configuration file. An example of its default contents follows:

```
    <bean                                              id="dbPropertiesBuilder"
class="gr.forth.rdfsuite.services.db.DbPropertiesBuilder">
        <property name="protocol" value="jdbc:postgresql"/>

        <!-- host of database -->
        <property name="host" value="127.0.0.1"/>
        <!-- port of database -->
        <property name="port" value="5432"/>
        <!-- database name -->
        <property name="dbName" value="caspar_hyb"/>
        <!-- username for the database -->
        <property name="username" value="username"/>
        <!-- password for the database -->
```

```
        <property name="password" value=""/>
        <property name="representation">
            <bean class="gr.forth.rdfsuite.services.db.DBRepresentation"
                  factory-method="valueOf">
                <!-- RDF Representation scheme in database. Available options:
[ISA, NOISA, HYBRID] -->
                <constructor-arg value="HYBRID"/>
            </bean>
        </property>
    </bean>

    <bean id="dbProperties" class="gr.forth.rdfsuite.services.db.DbProperties"
          factory-bean="dbPropertiesBuilder"
          factory-method="build"/>

     <bean id="ruqlConnector"
          factory-bean="dbProperties"
          factory-method="getRuqlConnector">
        <!-- Path of ICL executable -->
        <constructor-arg     index="0"     value="/home/swkm/services/ruql/hyb/rul-
1.0/bin/icl"/>
        <!-- Path of DBC executable -->
        <constructor-arg     index="1"     value="/home/swkm/services/ruql/hyb/rul-
1.0/bin/dbc"/>
        </bean>

    <bean id="ruqlConnections" class="java.lang.Integer"
          factory-method="valueOf">
        <!-- Initial pool of RUQL connections -->
        <constructor-arg value="5"/>
    </bean>

    <bean id="ruqlClient"
          factory-bean="dbProperties"
          factory-method="getRuqlClient">
        <!-- RUQL server port -->
        <constructor-arg index="0" value="2223"/>
        <constructor-arg index="1" ref="ruqlConnections"/>
    </bean>
```

Using this configuration file you can configure parameters like the database name and kind of stored representation, the path to the local executables of the RUQL server (it's assumed that the RUQL server was configured in the Hybrid mode and installed under the **/home/swkm/services/ruql/hyb** folder; you have to change these parameters accordingly to your installation path), usernames and passwords,etc.. You have also to have installed the RUQL server on the same machine as the application server.

Usually only the bold-faced values might need to be changed. A note of caution should be made here and concerns the fact that after doing any changes the web application needs to be redeployed, either by undeploying
and redeploying or by restarting the application server / servlet container.

8.5 USING THE SWKM SERVICES

The SWKM web services can be accessed through a standard WSDL interfaces. The corresponding WSDL descriptions of SWKM services can be retrieved either from the above mentioned web site or by accessing the services themselves after deployment like:

http://your.server.ip/CASPAR_SWKM_WS/ServiceName?wsdl

or from one of the currently working installations like:

http://139.91.183.8:3027/SwkmMiddlewareWS/

The available services are: importer, exporter, query and update and their full specifications were given in the previous sections. In addition, Gap Manager works on that installation.

8.5.1 A SWKM services' sample client

A sample client (SwkmClient.zip) has been provided (located in the same directory as the WAR file) that demonstrate the use of the corresponding SWKM web services

This is provided as a help to the user and a demo utility of the services' capabilities, any other SOAP enabled consumer technology or utility can be used.

In order to use the client, the contents of the file can be extracted to any folder. Apache ANT should be installed, version 1.6.5 or higher, which you can get here: http://ant.apache.org/. The file "services.properties" might be modified in order to contain the address of the web application and then "ant jar" should be executed. This will produce a dist(ribution) folder, which contains "SwkmClient.jar" and which is the actual client, and includes a folder "lib" with every needed jar (that should be placed on the Java CLASSPATH environment variable for the client to work).

8.6 A POSSIBLE DEPLOYMENT

Figure 8-1 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.

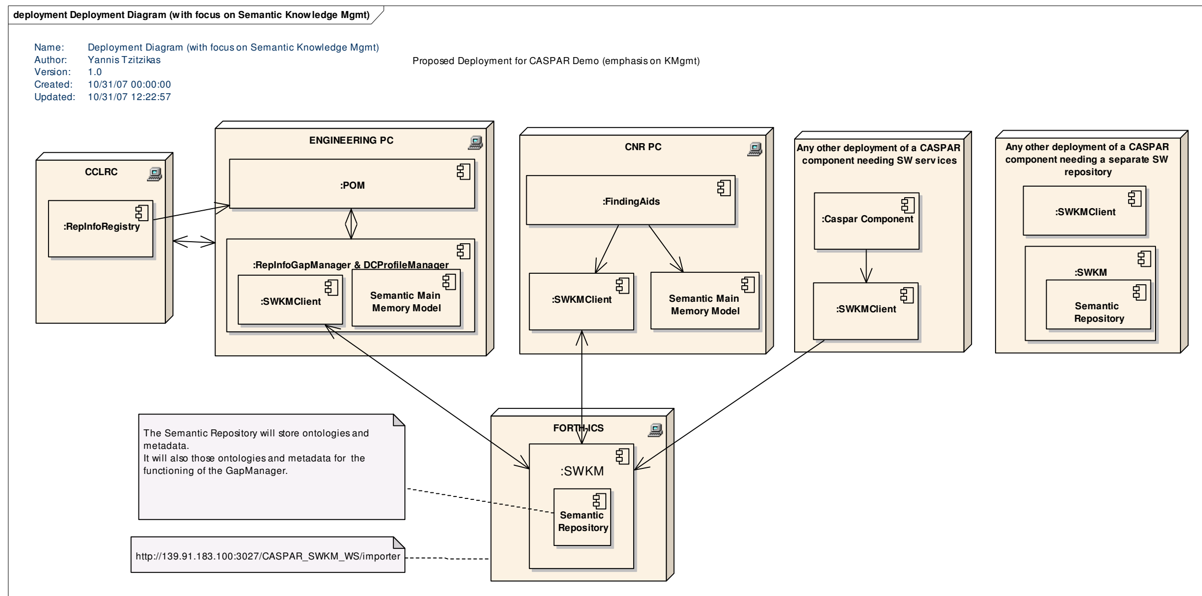


Figure 8-1 Proposed deployment for KM-related components

8.7 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):
< → <
> → >
" → "
' → '

& → &

9 OTHER USEFUL TOOLS

9.1 LIST OF TOOLS

Ontology Editors		
Protégé		
MISC		
XML2RDF converters		
DROID	Tool for automatic format identification	
Are there tools that could be used to automatically extract the dependencies ?	(one source is certainly the RepInfoRegistry)	
PROJECTS		
LUPA project	http://www.ics.forth.gr/isl/projects/projects_individual-gr.jsp?ProjectID=45 http://www.ubi-erat-lupa.org/	

9.2 USING PROTÉGÉ AND OTHER EDITORS

In order to be opened by Protégé, the schema files expressed in RDF/XML should use the RDF element rdf:about instead of rdf:ID.

Also, for a schema instantiation (in RDF/XML format) created using Protégé to be used for further processing with RDFSuite tools it is needed to remove RDF label elements and also delete the potential XML CDATA elements but preserves the content of the CDATA elements

This kind of automatic conversion is supported by a “Transformer” application which has as goal to assure the compatibility between files created/processed with Protégé and RDFSuite.

10 EXAMPLE OF RDF/XML AND TRIG FILE FORMATS

```
<?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/guernic
        <cult:technique>oil on canvas</cult:technique>
        <cult:exhibited>
```

dependencies

schema

data:only one UNNAMED graph

The RDF/XML file format

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfsuite: <file:/home/karvoun/downloads/demo_examples/rdfsuite.rdfs#> .
@prefix ms: <file:/home/karvoun/downloads/demo_examples/metaschema.rdf#> .
@prefix adm: <file:/home/karvoun/downloads/demo_examples/admin_baseuri.rdf#>
.
@bprefix                                                              :
<file:/home/karvoun/downloads/demo_examples/culture_schema_data.trig#> .

#default graph
    {
        #Classes
        :Artist rdf:type ms:RealWorldObject .
        :C2 rdf:type rdfsuite:Graph .
        :Artifact rdf:type ms:RealWorldObject .

        #Properties
        :graph_property2 rdf:type rdf:Property .
        :graph_property2 rdfs:domain :C2 .
        :graph_property2 rdfs:range xsd:string .
    }

:Museums {
   :rodin_museum rdf:type :Museum .
   :rodin_museum rdf:type adm:ExtResource .
   :rodin_museum adm:title "Rodin Museum"@en .
}

:Sculptures {
   :crucifixion rdf:type :Sculpture  .
   :crucifixion rdf:type adm:ExtResource .
   :crucifixion adm:title "Crucifixion"@en .
   :crucifixion :exhibited :rodin_museum .
   :Museums rdf:type :C2 .
   :Museums :graph_property2 "MUSEUMS"@en .
```

dependencies

schema

data:manyNAMED or UNMANED graphs

The TRIG file format