Deliverable No. 7.3

Hypermodels annotation services

Grant Agreement No.:	600841
Deliverable No.:	D7.3
Deliverable Name:	Hypermodels annotation services
Contractual Submission Date:	31/03/2016
Actual Submission Date:	15/04/2016

Dissemination Level		
PU	Public	
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	RE
CO	Confidential, only for members of the consortium (including the Commission Services)	

COVER AND CONTROL PAGE OF DOCUMENT	
Project Acronym:	**CHIC**
Project Full Name:	Computational Horizons In Cancer (CHIC): Developing Meta- and Hyper-Multiscale Models and Repositories for In Silico Oncology
Deliverable No.:	D7.3
Document name:	Hypermodels annotation services
Nature (R, P, D, O)[1]	R, P
Dissemination Level (PU, PP, RE, CO)[2]	RE
Version:	0.5
Actual Submission Date:	15/04/2016
Editor: Institution: E-Mail:	Pierre Grenon UCL p.grenon@ucl.ac.uk

ABSTRACT:

This deliverable documents the initial version of the CHIC semantic services dedicated to serve CHIC components centred around the management of hypermodels.

KEYWORD LIST:

CHIC semantic services, hypermodels

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n^o 600841.

[1] **R**=Report, **P**=Prototype, **D**=Demonstrator, **O**=Other

[2] **PU**=Public, **PP**=Restricted to other programme participants (including the Commission Services), **RE**=Restricted to a group specified by the consortium (including the Commission Services), **CO**=Confidential, only for members of the consortium (including the Commission Services)

MODIFICATION CONTROL			
Version	**Date**	**Status**	**Author**
0.1	31/03/16	Draft	GP
0.2	05/04/16	Draft	GP
0.3	06/04/16	Draft	GP
0.4	13/04/16	Draft	GP, NR, SS, TN
0.5	13/04/16	Draft	BdB, DD, SS, SG
0.6	14/04/16	Draft	GP, NR, SS, TN
0.7	15/04/16	Draft	BP, GP, NR, SS, TN
1.0	15/04/16	Revision	GP, NR, SS, TN

List of contributors

- GP: Pierre Grenon, UCL
- SS: Stelios Sfakianakis, FORTH
- TN: Nikolaos Tousert, ICCS-NTUA
- NR: Roman Niklaus, UBERN
- BdB: Bernard de Bono, UCL
- SG: Georgios Stamatakos, ICCS-NTUA
- DD: Dimitra Dionysiou, ICCS-NTUA
- BP: Philippe Buechler, UBERN

Contents

FIGURES

1 Executive Summary

In the present document, we provide a brief description of the semantic infrastructure followed by:

- a description of deployable services and technical documentation
- a documentation of the low level services currently supported
- a documentation of the mechanism used for extending these services
- a documentation of illustrative higher level services geared towards the Hypermodelling editor and Model repository

The contribution presented here makes provision for the evolution, modification and update of the knowledge representation used in support of semantic services and of the definition of targeted, specific service use and calls.

2 Introduction

2.1 *Purpose of this document*

The present document reports on the CHIC semantic services designed and deployed in support of semantic model and data management. The services described here will evolve consistently with any requirement arising from the client components in the CHIC architecture. The present account is based on requirements and scenarios of uses gathered from three main components, namely, the hypermodelling editor, the model repository and the clinical data repository. Throughout their prospective evolution, the documentation will be updated accordingly and additions and changes will be formally tracked through the use of a publicly available repository.

The intended scope of the present report is initially to support the use of semantic metadata and annotation of hypermodels. The scope is however broadly construed so as to register additional requirements for the use of semantic metadata within CHIC. In particular, higher-level services adapted to the Clinical Data Repository are added to the present considerations.

The rest of this introductory section provides a bird-s eye view on semantic annotation services within CHIC. Then requirements and scenarios of use are presented. Further sections address these requirements with a description of the conceptual representation of the resources to which semantic metadata is applied, the technical documentation for each infrastructure component supporting the semantic services is provided, and finally a technical description of high-level services as they are defined in their initial state.

2.2 *Semantic infrastructure for services*

The overall architecture can be abstracted to a very simple picture, Figure 1. In CHIC, a number of software components require the handling of semantic metadata through operations such as: the creation of metadata records and their storage, the querying of such records and the deletion of obsolete records. Services are designed to simplify these operations and mediate between clients and storage.

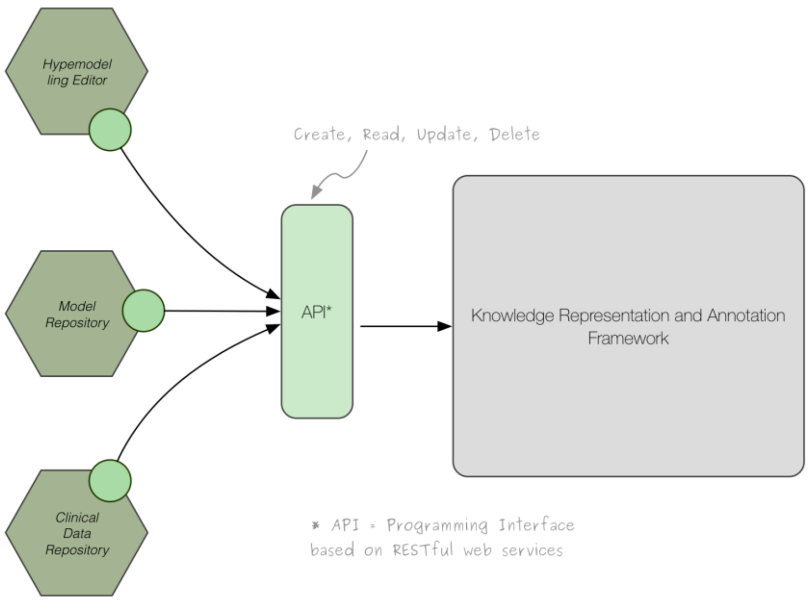

Hypemodelling Editor
Model Repository
Clinical Data Repository
Create, Read, Update, Delete
API*
Knowledge Representation and Annotation Framework
* API - Programming Interface based on RESTful web services

Figure 1 Simplified Architectural Overview: the semantic services serve CHIC clients with various abilities to connect to the semantic infrastructure

Services, of course, can be more or less generic and therefore more or less relevant or dedicated to different components. There are three main components in CHIC that are envisioned to be primary consumers of the semantic services:

1. Hypermodelling editor
2. Model repository
3. Clinical data repository

As each component caters for the management of different resources (models or clinical data objects, for example), the components may perform or require distinct specific operations. For example, operations dedicated to the annotation of a model parameter may be used by the Hypermodelling Editor but not by the Clinical Data Repository. Also, there may be selective differences between components regarding the range of generic operations they are brought to perform among the following main modes:

- Create – whereby a record is made in the relevant store
- Read – whereby a record is accessed through a query to the relevant store
- Delete – whereby a record is removed from the relevant store

Figure 2 sketches the envisioned operations for the Hypermodelling Editor as a way of illustration.

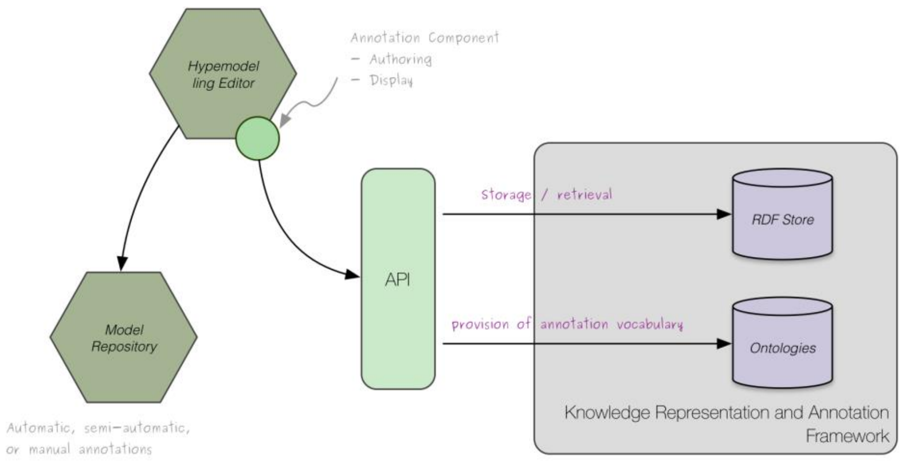

Hypemodel ling Editor
Annotation Component
– Authoring
– Display
API
Storage / retrieval
RDF Store
provision of annotation vocabulary
Ontologies
Knowledge Representation and Annotation Framework
Model Repository
Automatic, semi-automatic, or manual annotations

Figure 2 Prototypical Architectural Set up: links between the metadata infrastructure and the hypermodel editor and position of semantic services

There are, in CHIC, three main kinds of services and they type to two more specific parts of the semantic metadata infrastructure, Figure 3:

- Semantic Metadata services dedicated to interactions with the RDF store in which metadata statements about CHIC objects are recorded (in particular, hypomodels and hypermodels and their parameters and clinical data objects)

- Semantic reasoning services dedicated to interactions with the OWL Knowledge Base in which the ontologies that provide the formal domain vocabularies for making metadata statements are maintained (in particular, medical and anatomical ontologies)

- Terminology service, which, as an addition to the semantic reasoning services, provide a simpler and straightforward interaction with the human readable labelling of ontology concepts in the form of a basic terminology server

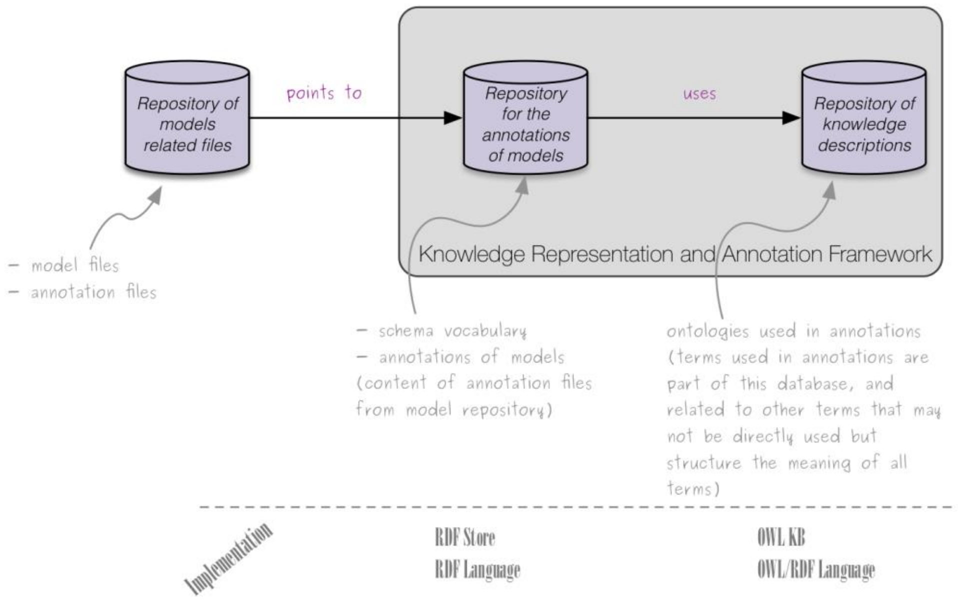

Repository of models related files
points to
Repository for the annotations of models
uses
Repository of knowledge descriptions
Knowledge Representation and Annotation Framework
– model files
– annotation files
– schema vocabulary
– annotations of models
(content of annotation files from model repository)
ontologies used in annotations (terms used in annotations are part of this database, and related to other terms that may not be directly used but structure the meaning of all terms)
Implementation
RDF Store
RDF Language
OWL KB
OWL/RDF Language

Figure 3 CHIC Semantic Metadata Infrastructure

3 Semantic services and infrastructure requirements

3.1 *Taking advantage of the Semantic Information in the Hypermodelling Editor*

3.1.1 Introduction

The editor is the end user facing application for the *design* of new hyper models in a visual and graphical way. The editor presents a “box and arrows” representation of the hypermodel, where each hypomodel is shown as a rectangle and each data exchange link between two hypomodels is depicted as line connecting them. The following picture shows the editor and an exemplary hyper model:

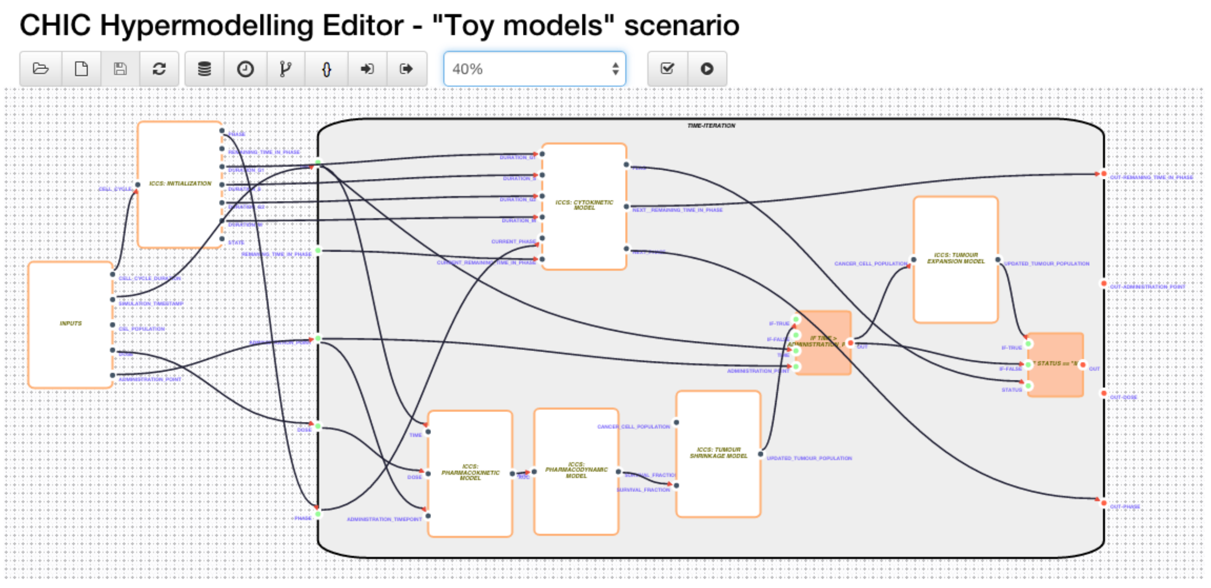


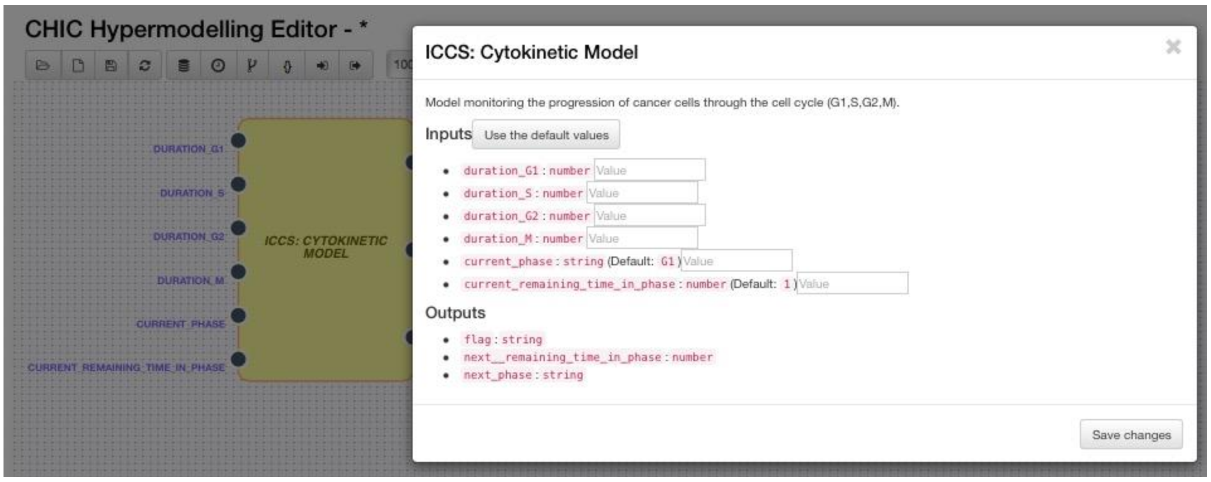
Figure 4 An example of a hypemodel designed in the CHIC Hypemodelling Editor

The editor allows the user to view basic metadata information for each model. As illustrated in Figure 5, this information includes:

- the **title** of the model
- the **description** of the model

and, for each in put or output parameter of a model:

- the **name** of the parameter,
- the **data type** for the parameter,
- and the **default value** of the parameter.



ICCS: Cytokinetic Model

Model monitoring the progression of cancer cells through the cell cycle (G1,S,G2,M).

Inputs Use the default values

- `duration_G1` : `number` Value
- `duration_S` : `number` Value
- `duration_G2` : `number` Value
- `duration_M` : `number` Value
- `current_phase` : `string` (Default: `G1`) Value
- `current_remaining_time_in_phase` : `number` (Default: `1`) Value

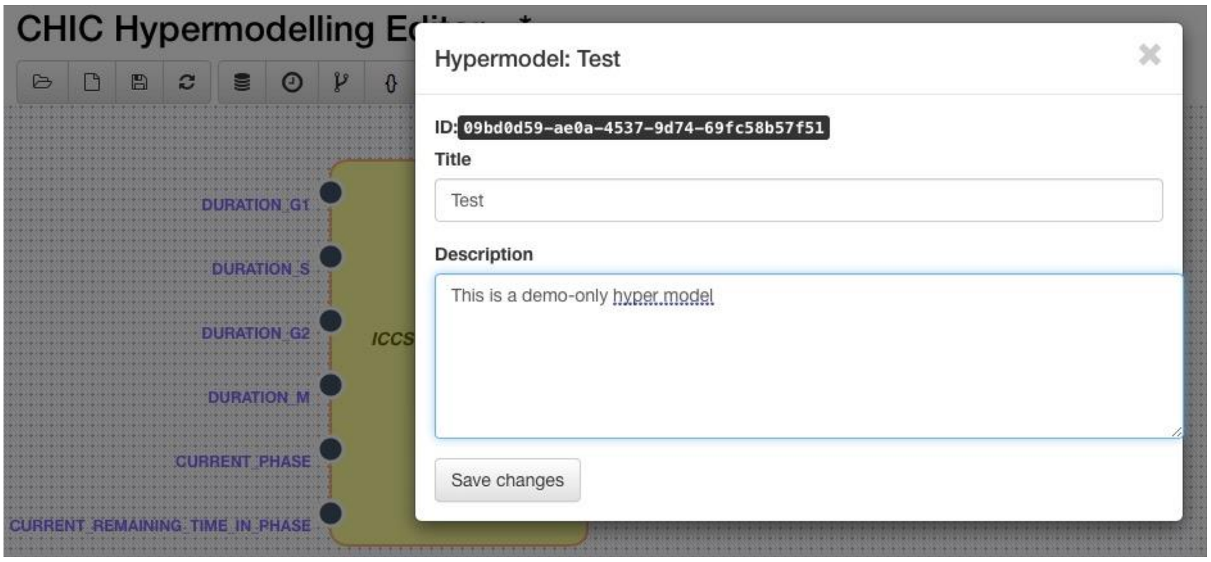
Outputs

- `flag` : `string`
- `next__remaining_time_in_phase` : `number`
- `next_phase` : `string`

Save changes

Figure 5 Information retrieved from the CHIC Model Repository for each hypomodel

Therefore, the editor is a consumer of model specific annotations and metadata information. On the other hand, it is also a producer of hyper model annotations for the newly constructed hyper models. Up until now, the metadata for new hyper models created in the Editor are just the title and description:


CHIC Hypermodelling Editor - *
DURATION_G1
DURATION_S
DURATION_G2
DURATION_M
CURRENT_PHASE
CURRENT_REMAINING_TIME_IN_PHASE
ICCS

Hypermodel: Test

ID: `09bd0d59-ae0a-4537-9d74-69fc58b57f51`

Title

Test

Description

This is a demo-only hyper model

Save changes

Figure 6 Information entered by the user during the registration of a new hypemodel

In addition to the name (title) and the description information, the final version of the editor will provide the following annotation properties for the new hypermodels:

- Inputs (multiple)
- Outputs (multiple)
- Author ("creator") information
- Publications related information
- Domain specific metadata, like temporal and spatial scale information
- Versioning information, with links to previous versions (the "parent" version that the current one was based on)

- Usage and runtime information, e.g. number of times executed, trial context information for each run, etc.

A simplified (and incomplete view) of the underlying resource oriented model of the editor is shown below:

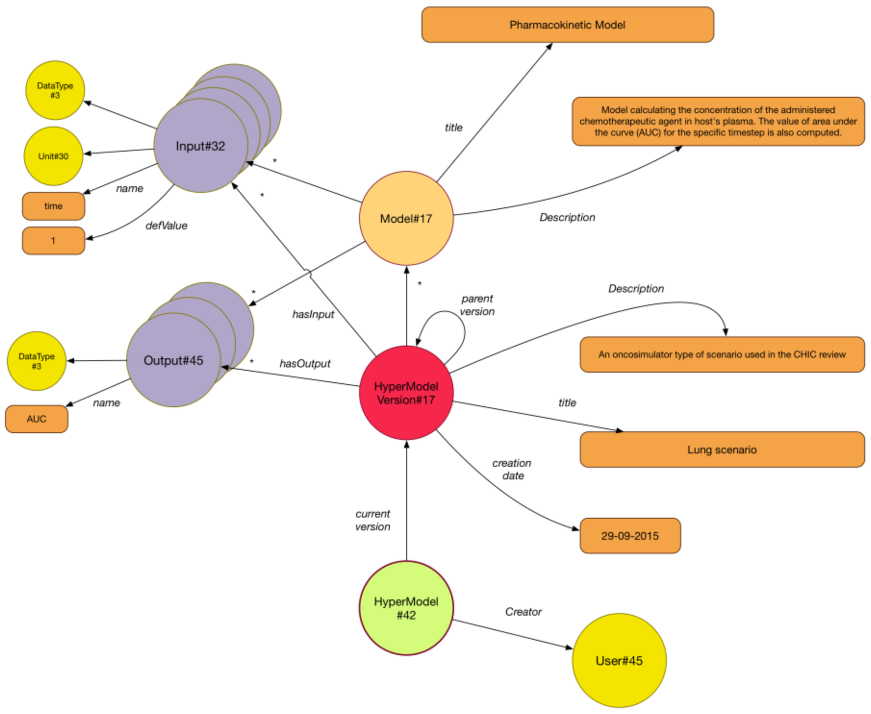


Figure 7 A generic schema for the information managed (produced and consumed) by the Hypermodelling Editor. The entities shown with circles represent "Resources", identified with URIs, in the sense of Resource Description Framework (RDF), while the rectangles represent "literals".

Here we concern ourselves with the possible ways that semantics can be exploited in the hypermodelling editor to facilitate the creation of new hypermodels. The following paragraph presents the usage scenarios that utilize semantic annotations of models towards this aim.

3.1.2 Use of semantics in the Editor

The semantic annotation of hypo models can be instrumental in the development of new hypermodels since these annotations can guide the users to the best way they can combine them, or to have a coarse categorization of them based on domain specific similarity criteria. Schulz et al [4] present a range of possibilities for taking advantage of the semantic information attached to computational models and data, as shown in the Figure below:

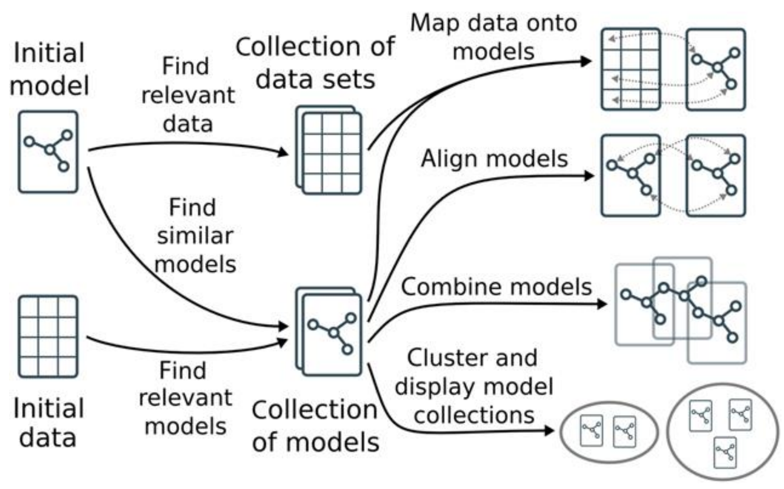


Figure 8 Model selection, alignment, and combination using Semantics (Schulz et al, 2011)

For the CHIC Hypermodelling Editor we envision two main areas that the semantics can play a significant role, in the so called Reactive and Proactive usage scenarios:

- On being Reactive: The Editor *reacts* to the actions of the users in order to check and validate them. It uses the semantic payload of the models and tries to check, for example, whether two hypomodels can be used together in a hypermodel or if they can even be linked by a connection so that the output of one can be given as input to the other.
- On being Proactive: The editor gathers information about the user and the goal s/he wants to achieve during the design of a hypermodel. It may be the case that it can provide *recommendations* in a proactive way by intelligent "guessing" of the next hypo models to put in the drawing canvas based either on the models already selected by the user him/herself or according to the similarities to the hypermodels designed by other users.

The Editor leverages the semantics that escort the models in order to support these Reactive and Proactive behaviours. In the following we present concrete examples for these intelligent behaviours.

3.1.2.1 Examples for Reactive behaviour: Consistency checking of the connections

The connections between hypomodels represent exchange of data, either as a stream of values in the case of "dynamic" inputs and outputs, or as a single value produced by a hypomodel before its termination and used by another one when commencing its execution. The consistency check of these links can use information about the **Units** used (e.g. for time, or volume), the **Data type** (e.g. integer, Boolean flags), the "**semantic type**" that provides a domain specific meaning for the exchanged information (e.g. "cell cycle duration"), etc.

Semantics can play a role both as facilitators for using a common terminology, as in the case of units, and as a translation and semantic linking facility for providing inference and high level matching of the used terms, as in the case of the "semantic type".

3.1.2.2 Examples for Proactive behaviour: Model recommendations

The "interface" (i.e. inputs and outputs) of (hypo) models can be also used to provide suggestions on what models to be used next. For example, the data types and semantic type of a model's outputs can be used as filters for the models that can be linked to. Therefore, the Editor can filter the list of possible models that can be selected and put in the hypermodel. Similar functionality can be supported when the semantic annotations of the models with respect to the "13 perspectives" (See Deliverable D6.1, [1]) are taken into account. For example, the "spatial scale" (Perspective II) provides a characterization for the level (atomic, molecular, cellular, etc.) that a hypo model operates on. Based on this information, and possibly the structure of existing hypermodels of the same or different users, the Editor can again propose a limited list of hypomodels for the user to include in the current hypermodel. Of course, in any case, these will be suggestions and not strictly enforced decisions on how the hypermodel should be built.

3.2 Model and Tool Repository makes use of the CHIC annotation services

The model and tool repository is one of the CHIC components that makes use of the higher level services, which are provided by the CHIC semantic infrastructure. Currently, information related to models and tools (descriptive information, information about parameters, etc.) is stored in a relational MySQL database. The use of a relational database for the storage of meta-information related to the models and tools provides the following benefits:

- SQL (Structured Query Language) databases use long-established standard, which is being adopted by ANSI & ISO.
- SQL queries can be used to retrieve large amounts of records from a database quickly and efficiently.
- Using standard SQL it is easier to manage database systems without having to write substantial amount of code.
- SQL is a complete language for a database and it is used to create databases, manage security of a database, update, retrieve and share data with users.
- SQL is used for linking front end computers and back end databases. Thus, it provides a client server architecture.
- SQL supports the latest object based programming.
- SQL is the database language which is used by businesses and enterprises throughout the globe.

Even though there are many reasons for using a relational database, the meta-information related to models and tools will be converted to RDF triples so as to be stored in the CHIC triplestore. RDF triples can be applied equally to all structured, semi-structured and unstructured content. By defining new types and predicates, it is possible to create more expressive vocabularies within RDF in order to describe information related to models. This expressiveness enables RDF to define controlled vocabularies with exact semantics. These features make RDF a powerful data model and language for data federation and interoperability across disparate datasets within CHIC.

The main advantages of RDF are the following [3]:

- **It is standard, open and expressive.** RDF is a standard model for data interchange on the web. RDF extends the linking structure of the web to use URIs to name the relationship between things as well as the two ends of the link (this is usually referred to as a “triple”).
- **RDF provides data interoperability.** Via various processors or extractors, RDF can capture and convey the metadata of information in unstructured (text), semi-structured (html, documents) or structured sources (standard databases). This makes RDF almost a “universal solvent” for representing data structure.
- **Schema unbound.** The single failure of data integration since the inception of information technologies - for more than 30 years, now – has been schema rigidity or schema fragility. That is, once data relationships are set, they remain so and cannot easily be changed in conventional data management systems or in the applications that use them. RDF has no such limitations. RDF is well suited and can provide a common framework to represent both instance data and the structures or schema that describe them, from basic data records to entire domains or world views.

- **Potentiality to increment, evolve, extend and adapt.** Indeed the very fluidity of RDF and structures based on it is another key strength. Since a basic RDF model can be processed even in the absence of more detailed information, input data and basic inferences can proceed early and logically as a simple fact basis. This strength means that either data or schema may be ingested and then extended in an incremental or partial manner. Partial representations can be incorporated and schema can extend and evolve as new structure is discovered or encountered.

Based on the current implementation, the user is able, among other things, to store a new model, categorize the model based on the thirteen perspectives [1] that have been defined within CHIC, and store information about the parameters of the model. All this information that the user submits through the graphical user interface, is converted and then stored in the model and tool repository's relational MySQL database. The design of the aforementioned database has been described in the deliverable "D8.1: Design of the CHIC repositories" [2]. For completeness reasons, the updated entity-relationship (ER) diagram of the model and tool repository is presented in Figure 9.

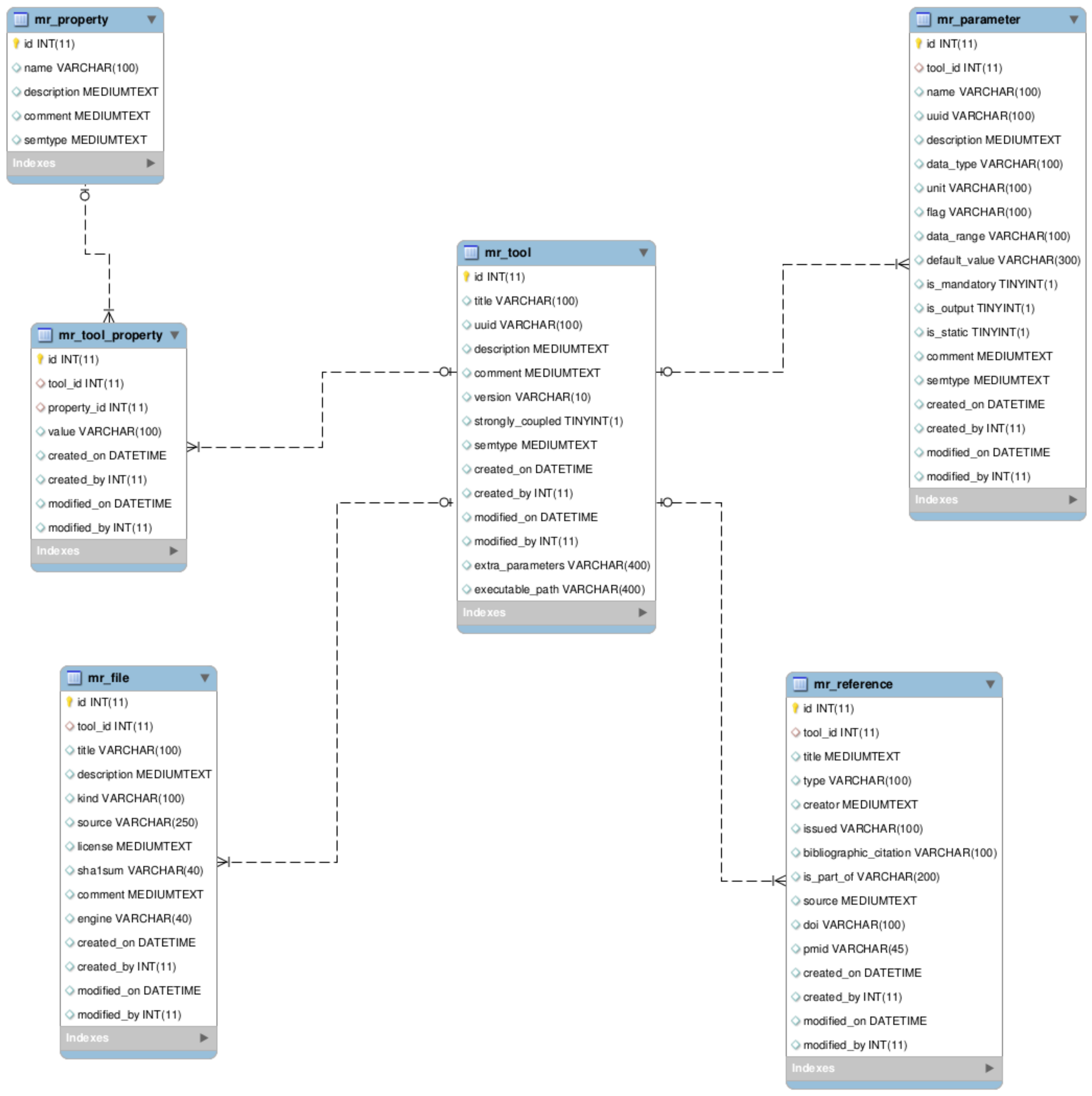
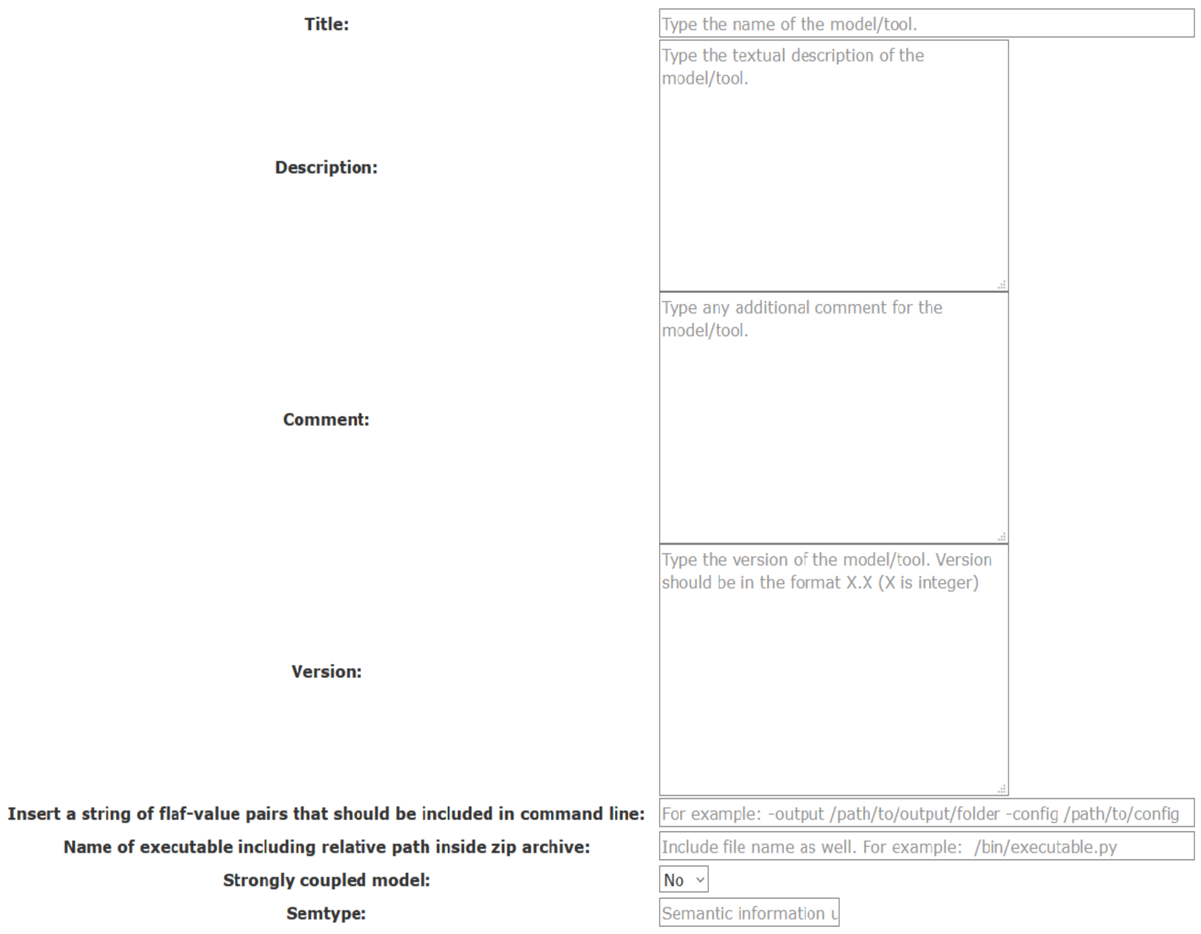


Figure 9 Entity relationship diagram of model and tool repository

As shown in Figure 9, the basic principles of the model and tool repository are the following [2] :

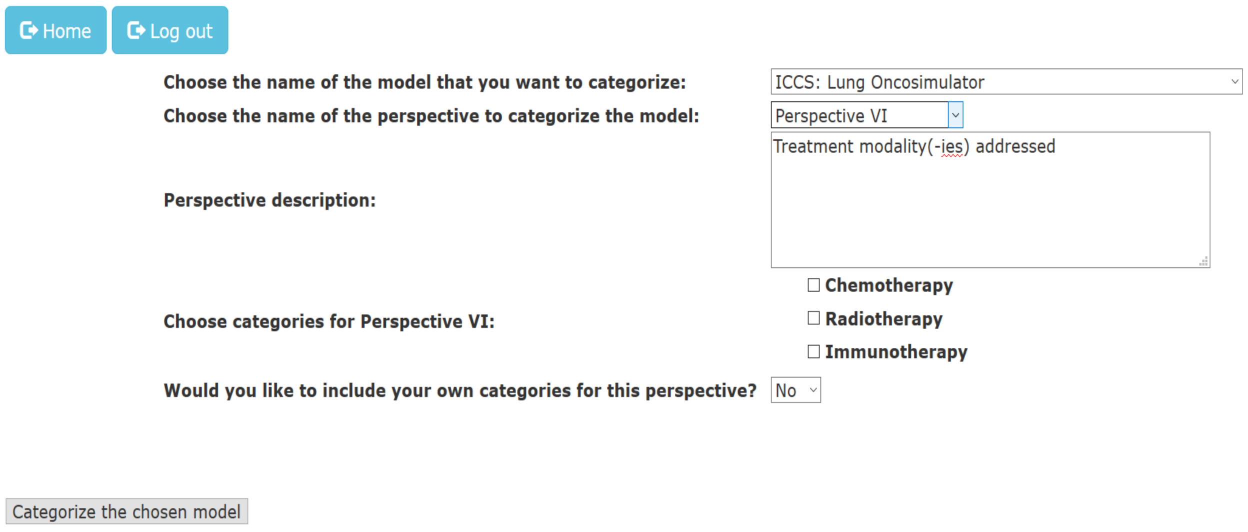
- Each model/tool has basic descriptive information, stored in the entity "mr_tool". This information uniquely defines the model/tool and differentiates it from other models/tools. The page where the user inserts basic descriptive information for their models is presented in Figure 10.
- Each model/tool can have one or more properties that further describes or/and classifies it. It must be noted that properties can only be used in correlation with mr_tool entity as they may only supplement the basic descriptive information of a model/tool. In CHIC project there is a set of predefined properties that reflect the 13 perspectives defined in "D6.1: Cancer hypomodelling and hypermodelling strategies and initial component models" [1].

- The descriptive information of properties is stored in the entity "mr_property". This entity does not contain the value of the property (related to a specific model/tool), but only the description of the property. The value that a property takes in case of a specific model/tool is stored in the entity "mr_tool_property".
- Entities "mr_property" and "mr_tool_property" assist the user in understanding the nature of the model/tool and facilitate the categorization of the models depending on the perspective from which they are viewed in the basic science context. The page where the user can categorize their models is presented in Figure 11.
- The models are treated as generic stubs, as described in "D7.1: Hypermodelling specifications", which have entry and exit points. Consequently each model/tool has various parameters, serving as input parameters or output parameters, which are stored in entity mr_parameter. This entity facilitates the transition from an abstract representation to a concrete one. The page where the user can create new parameters for their models is presented in Figure 12.



Title:	Type the name of the model/tool.
Description:	Type the textual description of the model/tool.
Comment:	Type any additional comment for the model/tool.
Version:	Type the version of the model/tool. Version should be in the format X.X (X is integer)
Insert a string of flaf-value pairs that should be included in command line:	For example: -output /path/to/output/folder -config /path/to/config
Name of executable including relative path inside zip archive:	Include file name as well. For example: /bin/executable.py
Strongly coupled model:	No
Semtype:	Semantic information u

Figure 10 The page where the user inserts basic descriptive information for their models

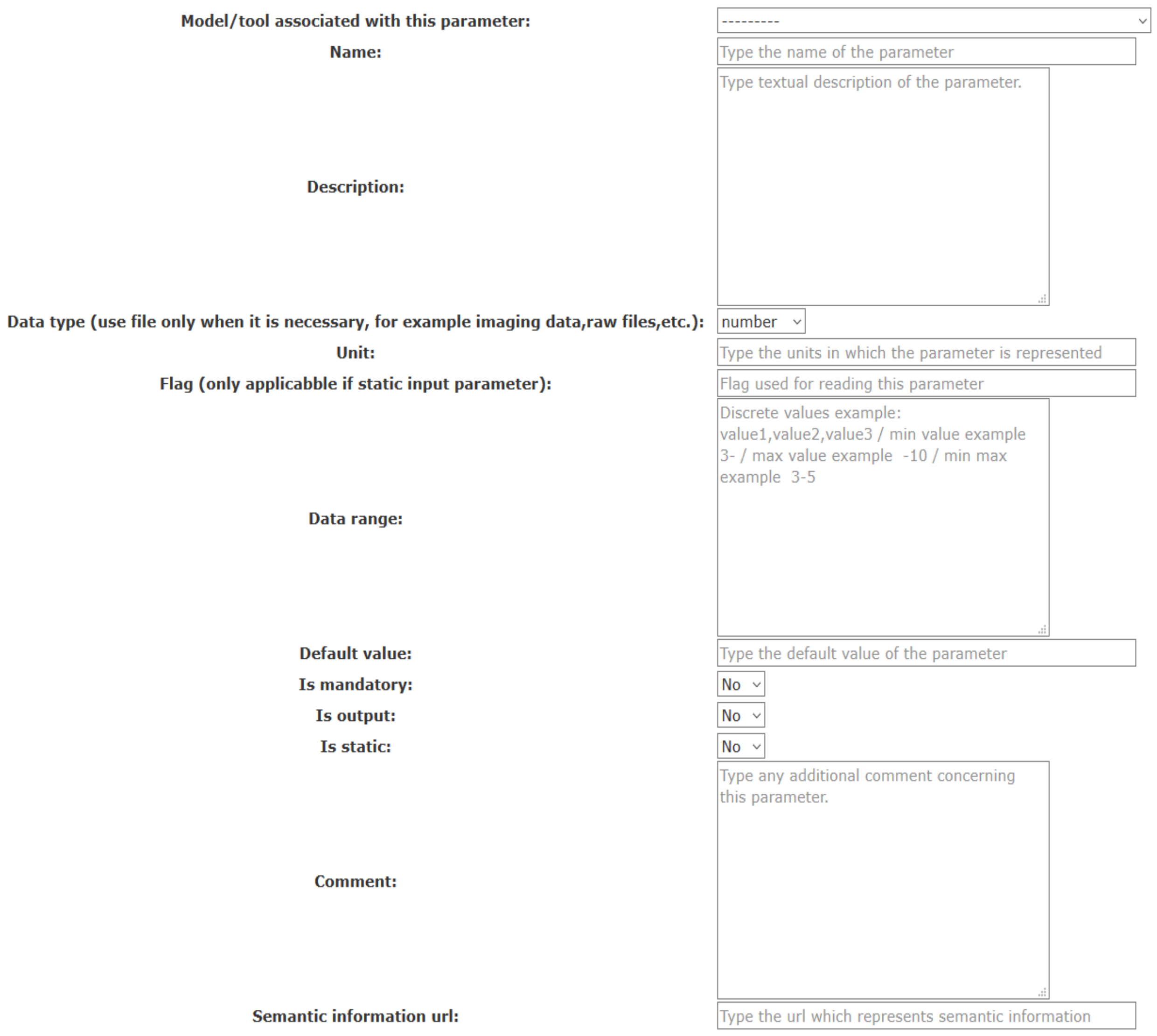


Home | Log out

Choose the name of the model that you want to categorize:	ICCS: Lung Oncosimulator
Choose the name of the perspective to categorize the model:	Perspective VI
Perspective description:	Treatment modality(-ies) addressed
Choose categories for Perspective VI:	☐ Chemotherapy ☐ Radiotherapy ☐ Immunotherapy
Would you like to include your own categories for this perspective?	No

Categorize the chosen model

Figure 11 The web page where the user categorizes their models



Model/tool associated with this parameter:	---------
Name:	Type the name of the parameter
Description:	Type textual description of the parameter.
Data type (use file only when it is necessary, for example imaging data,raw files,etc.):	number
Unit:	Type the units in which the parameter is represented
Flag (only applicabble if static input parameter):	Flag used for reading this parameter
Data range:	Discrete values example: value1,value2,value3 / min value example 3- / max value example -10 / min max example 3-5
Default value:	Type the default value of the parameter
Is mandatory:	No
Is output:	No
Is static:	No
Comment:	Type any additional comment concerning this parameter.
Semantic information url:	Type the url which represents semantic information

Figure 12 The page where the user can create new parameters for their models

As shown in Figures 10, 11 and 12, the user is prompted to submit meta-information for their models (model description, information about parameters, etc.) in order for the system to store persistently this information to the corresponding MySQL tables. Nevertheless, apart from the storage of the meta-information in the local relational database, the corresponding information is also going to be stored in the CHIC RDF store.

The model and tool repository and the CHIC semantics infrastructure will make use of a common RDF mapping configuration file so as to produce a model (a set of RDF triples) based on the already locally stored relational data. The aforementioned configuration file will map some of the model repository's database tables and columns to CHIC RDF vocabularies and OWL ontologies. Consequently, this mapping will define the virtual RDF graph that will contain some of the information from the model and tool repository's MySQL database.

It should be noted that not all the information stored in the model repository's MySQL database will be exposed to the CHIC RDF triplestore, but only part of it. The part of the model repository that needs to be transformed to RDF triples depends on the requirements from the perspective of the CHIC semantic web clients. Based on the relational database schema that has been presented in Figure 9, the following attributes are going to be used for the semantic annotation of the models. :

- Regarding the "mr_tool" entity, the "semtype" attribute will be mapped to the URI of the model and the content of "title" and "strongly_coupled" attributes will be mapped to the corresponding RDF property values.
- Regarding the "mr_parameter" entity, the "semtype" attribute will be mapped to the URI of the parameter and the content of "name", "data_type", "unit", "default_value", "is_mandatory", "is_output", and "is_static" attributes will be mapped to the corresponding RDF property values.
- Regarding the "mr_property" entity, the "semtype" attribute will be mapped to the URI of the perspective and the content of the "name" attribute will be mapped to the corresponding RDF property value.
- Regarding the "mr_tool_property" entity, the "value" attribute will be stored as an RDF property value.

The topology of the CHIC infrastructure that handles the semantic annotation of the models is presented in Figure 13. The following modules are used for the use case of the model semantic annotation:

- **Controller**: The controller is the central module of the model repository that consists of many other submodules. It opens the local relational database connection and it handles web requests and presentation details that the user will see. It also calls the Loader module.
- **Loader**: The Loader is in charge of converting MySQL data into RDF property values that will be provided to the CHIC semantics infrastructure web services. It also loads the RDF mapping configuration file and calls the application programming interfaces of the CHIC metadata store.
- **RDF mapping configuration file**: This file includes the necessary information for mapping MySQL table and columns of the CHIC model repository to RDF properties, vocabularies and OWL ontologies of the CHIC metadata store.
- **API**: This module consists of all the web annotation services that are exposed from the CHIC metadata store.

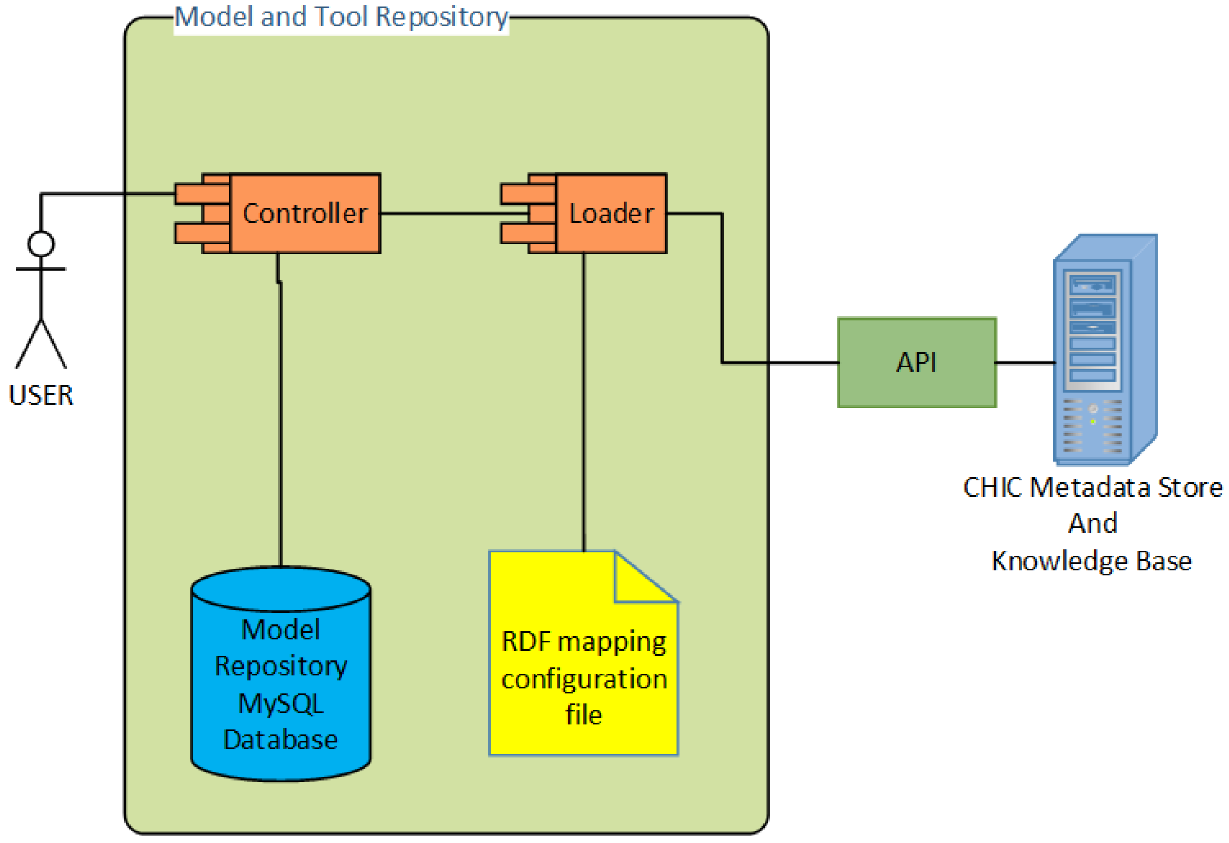


Figure 13 Topology of the CHIC infrastructure that handles the semantic annotation of the models

3.3 Clinical Data Repository

The clinical data repository will permanently host all the related medical data produced or collected by the CHIC project. The data will not be directly provided by the clinical environment. The data will pass through de-identification and (pseudo)-anonymization processes, as described in WP4. Additionally, interfaces that will allow to import and export the contents of the clinical data repository will be developed. In this way the data can be sustained after the expiration of the project's lifetime and reused and exploited continuously within the limits allowed by the legal framework of the project. The export services that will be created will also assist in this direction, as many of the data sets to be gathered by the CHIC project will be reusable by future projects. The clinical data repository contains all the relevant medical data including imaging data, clinical data and genetic / molecular data.

The concept underlying the design of the clinical data repository is to rely on generic objects. The chosen approach ensures the ability of the system to handle new type of objects with minimal additional efforts. However, even though the design is very generic, the structure of the data to be stored is used to provide appropriate verification of the consistence of the data and to extract all the relevant metadata associated with each uploaded object.

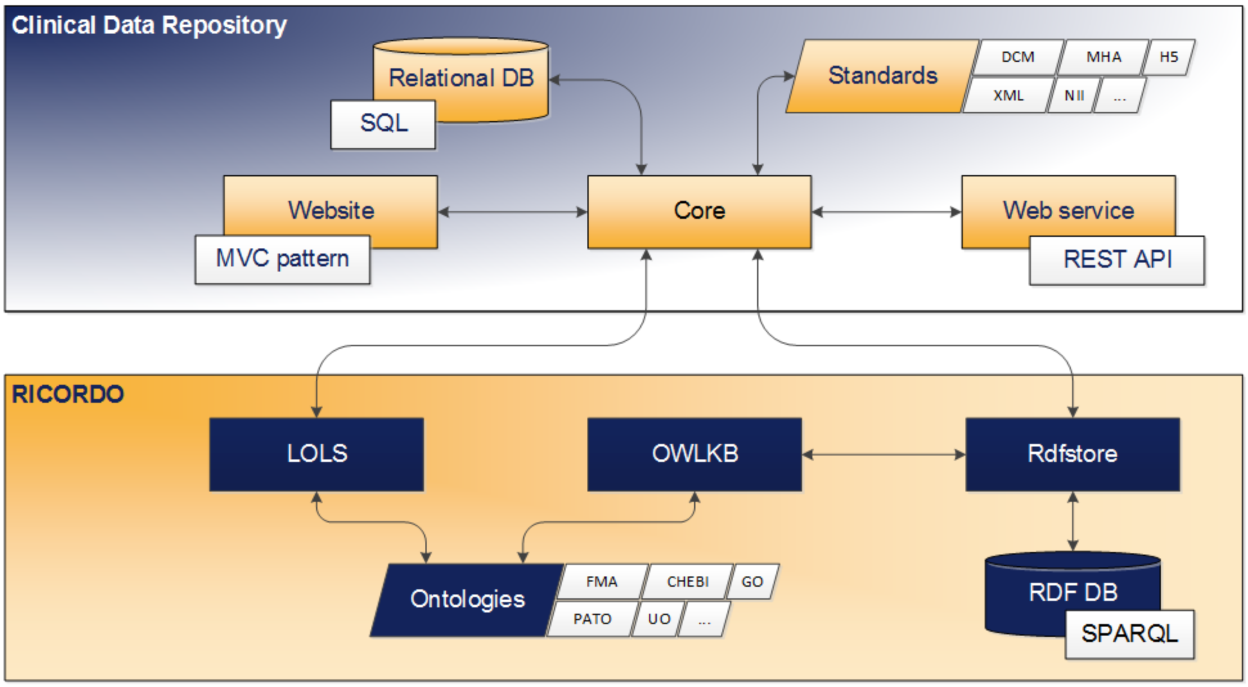


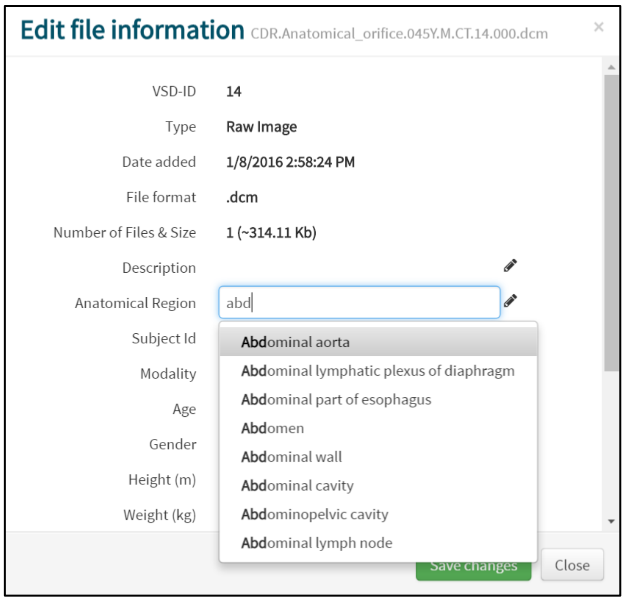
Figure 14 A schematic representation of interactions between Clinical Data Repository and RICORDO components.

The clinical data repository can be accessed either by website or by web service. The former is geared towards end users and the latter for third-party applications but both use the same core. The common core relies on a relational database which makes use of the Structured Query Language (SQL). Standard file formats as described in D8.1 [2] supported by the clinical data repository include DICOM, MetaImage, Analyze and Nifti for medical imaging data, CDISC ODM XML for clinical data and MINiML XML for genetic / molecular data. One objective is to extract selected metadata from the files during the upload process to the clinical data repository automatically. Another objective is to let users such as clinicians, researchers and others annotate the objects of the clinical data repository manually. Both objectives have in common that the annotations will be exported to the semantics infrastructure provided by RICORDO within CHIC.

RICORDO offers three components called LOLS, Rdfstore and OWLKB which are relevant within CHIC. The intended purpose of Local Ontology Lookup Service (LOLS) is to translate standardized (but not human readable) identifier strings used for triplestores to human-readable labels describing them for a given set of ontologies. Rdfstore is a metadata wrapper based on templates serving as a messenger between SPARQL endpoint and end-user, obviating the need to learn complicated SPARQL syntax. OWLKB is a semantic reasoner which enables to query semantic data loaded from an ontology. Both components LOLS and OWLKB have the same set of ontologies in common.

3.3.1 Interactions with the Local Ontology Lookup Service

In order to enable clinicians, researchers and others to annotate objects of the clinical data repository manually a connection with LOLS is needed. An exemplary use case is the annotation of an object with anatomical regions. As shown in Figure 15, the user starts to type the name of the anatomical region and the autocomplete function offered by LOLS returns a list of matching entries. The user selects the correct entry from the list which completes this step of the annotation process. In this case, it would not make sense to present matching entries other than those from the Foundational Model of Anatomy (FMA) ontology to the user. Therefore, the crucial functionality to filter the range of ontologies to be searched by the autocomplete function is required.


Edit file information CDR.Anatomical_orifice.045Y.M.CT.14.000.dcm

VSD-ID	14
Type	Raw Image
Date added	1/8/2016 2:58:24 PM
File format	.dcm
Number of Files & Size	1 (~314.11 Kb)
Description	
Anatomical Region	abd
Subject Id	
Modality	
Age	
Gender	
Height (m)	
Weight (kg)	

Figure 15 User dialog to annotate an object with anatomical regions using the autocomplete function offered by the Local Ontology Lookup Service (LOLS).

3.3.2 Interactions with the Rdfstore

Metadata can be extracted during the upload process by the clinical data repository automatically, if standard file formats are used. However, in the majority of cases the extracted metadata is not in the form to be stored directly in the Rdfstore. Therefore, the metadata must be processed to triples before being exported to the Rdfstore. This is one of the reasons the clinical data repository stores the extracted metadata in the relational database. Another reason is the export process itself which requires a reliable retry logic. Last but not least, the clinical data repository needs to be able to display the information associated with each object without fetching it from the Rdfstore every time. The Rdfstore itself already offers the functionality to add and delete triples in order to enable interactions with the clinical data repository.

Adding and deleting triples to/from the Rdfstore is merely a means to an end. The main objective is to leverage the powerful search capabilities offered by the nature of the semantic technology. For this purpose, the Rdfstore offers an extensible template system which can be used for querying. A simple query such as “get all objects having more than one file” can be achieved by the Rdfstore directly. Once the query involves information stored in an ontology such as “get all objects which are part of FMA Head" the Rdfstore relies on the semantic reasoner offered by OWLKB. A direct interaction between the clinical data repository and the OWLKB is not intended.

Figure 16 illustrates the planned relational database schema which extends the existing covered in deliverable D8.3. The selected architecture allows different scenarios. A possible scenario could be to support multiple triplestores concurrently. This means all triples could either be exported to all triplestores, a subset of triples could be exported to a subset of triplestores or a combination thereof. Another scenario could be to replace the existing triplestore just by re-initiating the export process.

Table name	**Table description**
TripleObject	Stores the value provided by the user or extracted during upload (e.g. ontology terms, literals, numbers, RDFS class IRIs, etc.).
Ontology	Stores the different supported ontology types (e.g. FMA, CHEBI, etc.).
TriplePredicate	Stores the different predicates (e.g. has_age, has_gender, etc.).
AnnotationTriple	Stores the information about the triple.
TripleSubject	Stores the IRI of the subject.
TripleSubjectType	Stores the type of the subject (e.g. object, file, user, etc.).
Object	The object is the basis element of the clinical data repository. It can be a medical image, clinical trial, genomic sample, etc.
File	This entity contains the information where exactly one binary file is stored on the clinical data repository file system. A file can be used by multiple objects and an object can have multiple files.
User	A user entity contains personal information of the user such as the first name and last name.
Triplestore	Stores the information about the different supported triplestores (e.g. Rdfstore provided by RICORDO)
AnnotationTripleLog	Stores information about background processes responsible for exporting triples to the triplestore.

Table 1: A description of all tables involved in the clinical data repository database schema extension illustrated in Figure 16.

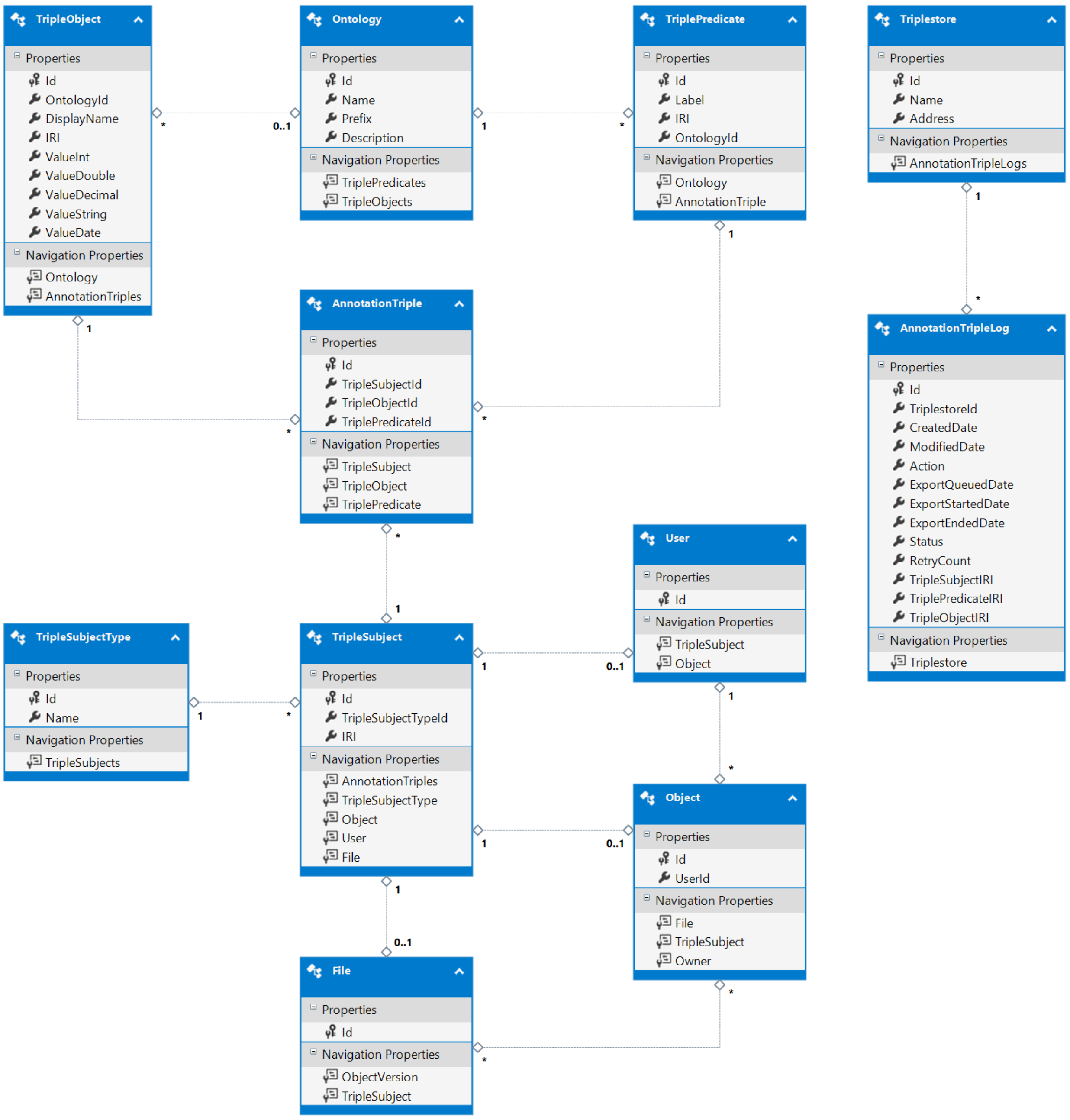


Figure 16 Extension of the clinical data repository database schema to persist triples, export and retry logic information.

4 Semantic Metadata

Semantic metadata is a form of knowledge representation whereby objects are represented through a (unique) identifier and a number of statements in a given representation language are used to record information or formal descriptions.

Outside of the scope of this section is the manner whereby objects so described are provided with a unique identifier. CHIC components producing metadata statement may have to provided for globally unique identifiers in the form of URI. Standardly, URIs are two parts IDs that contain a namespace element and an identifier element, so that at least the identifier is unique within the encompassing namespace.

4.1 Resource Descriptions Schema

We use a simple and pragmatic approach to the knowledge representation backbone supporting the semantic infrastructure. The range of objects that may be annotated and carry semantic content is represented schematically in the language of RDF, the Resource Description Framework. This schematic representation can also be called an ontology of the CHIC resources.

According to this view, each object of interest belongs to a class to which we can attach a number of annotational links to a semantic object (for example, an object representing the biological meaning of a parameter), another annotatable object (for example, linking a parameter to a model) or, in some cases, a value from a given data type (for example, a number, a string of text or a date). The links are relations between an annotated object on the one hand and a concept on the other. The nature of the linkage is elucidated by the intended semantic content of the link, i.e., its meaning. We do not overwork the formalisation of the meaning of links and treat them in general as (formally) undefined and (informally and conventionally) elucidated terms.

When applicable, the linked semantic objects belong to instrumental domain ontologies themselves. A domain ontology is a formalised theory of a domain, for example, anatomy or the domain of laboratory measurements. These ontologies are instrumental in relation to the theory of CHIC resources because they are used to provide values in the annotation of the latter. The ontologies used in the current stage of development are indicated in the next section.

4.1.1 CHIC Resource Ontology

CHIC resources handled by the hypermodelling editor and the model repository are few in kinds; they are either models (hyper or hypo) or parameters of models (input or output). The kind of resources handled by the Clinical Data Repository can be conceived according to two different dimensions. According to a first dimension, there are the internal clinical data resources which include users, files, and data objects. Each of these have various properties which are relevant to their articulations. According to a second dimension, the data objects themselves can be of several distinct types depending on the type of clinical data stored and the format of their storage (images, trial results, laboratory and experimental results).

We initialise a CHIC Resource Ontology with a small set of classes and relationships holding between instances of these classes. The goal of this ontology is to provide the formal tools for the representation of CHIC resources and provide the backbone for the definition of a number of attributive relationships allowing for the annotation of CHIC resources.

4.1.1.1 Categories of Objects

- Modelling objects – a class of objects. An instance of the class Modelling Object is a model or a part (logical or conceptual) of a model, in particular a parameter.

- Mathematical models – a subclass of the class of modelling objects. An instance of the class MathematicalModel is a model.
- Hypermodels – a subclass of the class of mathematical models. An instance of the class Hypermodel is a hypermodel in the sense of CHIC technical specifications.
- Hypomodels – a subclass of the class of mathematical models. An instance of the class Hypomodel is a hypomodel in the sense of CHIC technical specifications.
- Parameter of a mathematical model – a subclass of the class of modelling objects such that an instance of this class is a parameter of a mathematical model.
- Input parameter of a mathematical model – a subclass of the class of parameter of a mathematical model such that an instance of this class is an input parameter of some mathematical model.
- Output parameter of a mathematical model – a subclass of the class of parameter of a mathematical model such that an instance of this class is an output parameter of a mathematical model.

4.1.1.2 Relations between objects

We use three relations between models and their parameters:

- parameter-of to link generically a model to one of its parameters, or its converse
- input-parameter-of to link a parameter to a model when the parameter is an input parameter of the model
- output-parameter-of to link a parameter to a model when the parameter is an output parameter of the model

4.1.1.2.1 Illustration

A model, for example, the Wilms Tumour model developed in CHIC, is be represented by a specific URI, and so is everyone of its parameters. The above may be used to record minimal information about these and their link.

```
chicdb:model1
  rdfs:label "Wilms" ;
  rdfs:comment "Wilms model" ;
  rdf:type mo: Hypomodel .

chicdb:parameter1
  chicro:input-parameter-of <#model1> ;
  rdfs:label "Cell cycle duration of stem cells" .
```

Illustration of the use of the CHIC Resource Ontology to define resources, using RDF TTL syntax. “chicdb”, “rdfs”, “rdf” and “chicro” are namespace abbreviations for a database of annotations, the RDFSchema, RDF and the CHIC resource ontology namespaces, respectively.

Note that we make full use of the RDF standard for at least two things here, on the one hand, lexical information (using rdfs:label) and human readable descriptions (using rdfs:comment), and on the other hand, we use the inbuilt rdf:type to indicate the belonging of an instance, here, the model or the parameter, to a class.

4.1.1.3 Annotation relations

The CHIC Resource ontology is modifiable and extensible, we introduce here a few basic annotation relationship from which to prime the coverage of the requirements sketched above. This is where we list the relationships that are specifically used to enter semantic metadata information regarding CHIC resources.

4.1.1.3.1 Annotation of objects – irrespective of their subtype

Name	Symbol in CHICRO	Domain	Range
Title or Name	hasName	chicro:Object	rdfs:Literal
Description	hasDescription	chicro:Object	rdfs:Literal

4.1.1.3.2 Annotation of models

We define 13 annotation relationships corresponding to each of the intended perspectives (D6.1, [1])

Name	Symbol in CHICRO	Domain	Range
Perspective 1	hasPositionIn-01	chicro:Model	Set of relevant URIs for Tumour-Affected Normal Tissue Modelling
Perspective 2	hasPositionIn-02	chicro:Model	Set of relevant URIs for Spatial Scale(s) of the Manifestation of Life
Perspective 3	hasPositionIn-03	chicro:Model	Set of relevant URIs for Temporal Scale(s) of the Manifestation of Life
Perspective 4	hasPositionIn-04	chicro:Model	Set of relevant URIs for Biomechanism(S) Addressed

Perspective 5	hasPositionIn-05	chicro:Model	Set of relevant URIs for Tumour Type(S) Addressed
Perspective 6	hasPositionIn-06	chicro:Model	Set of relevant URIs for Treatment Modality(-ies) Addressed
Perspective 7	hasPositionIn-07	chicro:Model	Set of relevant URIs for Generic Cancer Biology – Clinically Driven Character of the Modelling Approach
Perspective 8	hasPositionIn-08	chicro:Model	Set of relevant URIs for Order of Addressing Different Spatial Scales
Perspective 9	hasPositionIn-09	chicro:Model	Set of relevant URIs for Order of Addressing Different Temporal Scales
Perspective 10	hasPositionIn-10	chicro:Model	Set of relevant URIs for Mechanistic-Statistical Character of the Modelling Approach
Perspective 11	hasPositionIn-11	chicro:Model	Set of relevant URIs for Deterministic-Stochastic Character of the Modelling Approach
Perspective 12	hasPositionIn-12	chicro:Model	Set of relevant URIs for Continuous-Finite-Discrete Character of the Mathematics Involved
Perspective 13	hasPositionIn-13	chicro:Model	Set of relevant URIs for Closed Form Solution – Algorithmic Simulation Modelling Approach

4.1.1.3.3 Annotation of parameters

Name	Symbol in CHICRO	Domain	Range
datatype	parameterhasDatatype	chicro:Parameter	xsd datatypes (and possible additions)
unit	parameterHasUnit	chicro:Object	Set of relevant URIs

4.1.1.3.4 Semantic typing

In order to record the interpretation of a given object according to a domain specific definition, for example, that a given parameter is a rate of cell killing, we use a specific relationship rather than the rdf:type property.

Name	Symbol in CHICRO	Domain	Range
Semantic type	hasInterpretedType	chicro:Object	Set of relevant URIs

4.2 Ontologies used

Whenever the annotation relationships introduced as part of the CHIC Resource Ontology refer to a set of URIs for filling in the information, the URIs in question are taken from domain specific ontologies.

In CHIC, we reuse third parties, externally maintained and standardised ontologies for certain domains and we also make provision for developing smaller, more adequate ontologies that fit the CHIC purpose more straightforwardly and adequately. As with the CHIC Resource Ontology, any ontological resource used here is also subject to modification and revision.

These ontologies are accessible programmatically within the semantic infrastructure through the OWL knowledge base which is a database server containing versions of the ontologies in the OWL language. If we so choose to change the ontologies used, we would be in a position to update the RDF annotation records using them, provided we maintain a suitable mapping.

From the above, we need ontological references for

i) Units of Measurements

ii) Perspectives 1 to 13

iii) Semantic types.

The ontological resources needed are unequal in complexity depending upon the sort of record we intend to maintain within the RDF annotation store. Here we keep a minimal list of the ontologies used for entering specific annotation-relations values and we sketch the complexity of the ontology ecosystem involved in filling in the semantic typing information.

Our strategy is to enable the use of semantic annotation and then to go through cycles of refinements. For this reason we will adopt the following:

Domain		
Units of measurement	Unit Ontology	http://obofoundry.org/ontology/uo.html
Perspectives	Existing Ontologies or Ad Hoc Ontology	Existing ontologies or parts of existing ontologies can be used for different perspectives (for example, the Disease Ontology for perspective 5 or the Unit Ontology for perspective 3) or ad hoc ontologies can be developed for others.
Semantic types	Combinations of ontologies, among which: FMA CHEBI GO PATO Ad Hoc Extensions	The external ontologies are developed and maintained within the framework of the Open Biomedical Ontologies (OBO Foundry, http://obofoundry.org/). Ad hoc extensions are the subject of further development and involve extensions and combinations of the above.

5 Semantic Infrastructure Supporting Services

The CHIC semantic infrastructure has been deployed on CHIC servers. The demonstration GUI for the Rdfstore 2.0 services is accessible from the following URL:

http://139.91.210.22:20081/gui

Within this deployed infrastructure the base host for the CHIC services is therefore: http://139.91.210.22:20081.

Similarly, the demonstration GUI for the OWLKB can be accessed here:

http://139.91.210.22:20080/gui

The demonstration GUI for the LOLS can be accessed here:

http://139.91.210.22:20084/gui

5.1 *Triple store and Rdfstore interface*

Rdfstore 2.0 is the so-called Ricordo metadata wrapper. It serves as a messenger between SPARQL endpoint and end-user. The motivation behind Rdfstore 2.0 is to alleviate the need to handle SPARQL syntax and make it simpler and more straightforward to deal with metadata. This is done with a system of templates, which are customized at the organizational level.

The end-user tells the appropriate systems team: ``I want a form that'll let me query the database for X.'' The team creates a template for that query. Now the end-user can select that template and query the database for X by filling out a simple form, no SPARQL required.

5.1.1 Installation Requirements

- Java and Java runtime: Rdfstore 2.0 requires Java Runtime and a Java compiler for its installation.
- RDF Triple Store: Rdfstore 2.0 assumes that a triple store is running and has exposed a SPAQRL endpoint. Rdfstore 2.0 has been used in combination with the following:
  - Virtuoso Open-source Edition (GNU GPL license)

    http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/
  - Fuseki and Fuseki2 (Apache 2.0 license)

    https://jena.apache.org/documentation/fuseki2/
  - Specific configuration information is provided below for these.

5.1.2 Installation Steps

1. Rdfstore 2.0 is available for download from:

   http://github.org/semitrivial/rdfstore

   Use "git clone", or any other means, to copy this repository locally.

2. Within the directory containing the copy of the repository, compile Rdfstore using "make" (or manually: "javac -g Rdfstore.java")

3. Within that directory, run "java Rdfstore -help" to get help on the command line arguments, or see further below.

In order for Rdfstore 2.0 to be useful, *templates* need to be present in the relevant subdirectory, as discussed below.

5.1.3 Running Rdfstore 2.0

Rdfstore 2.0 can be run with the following command-line arguments:

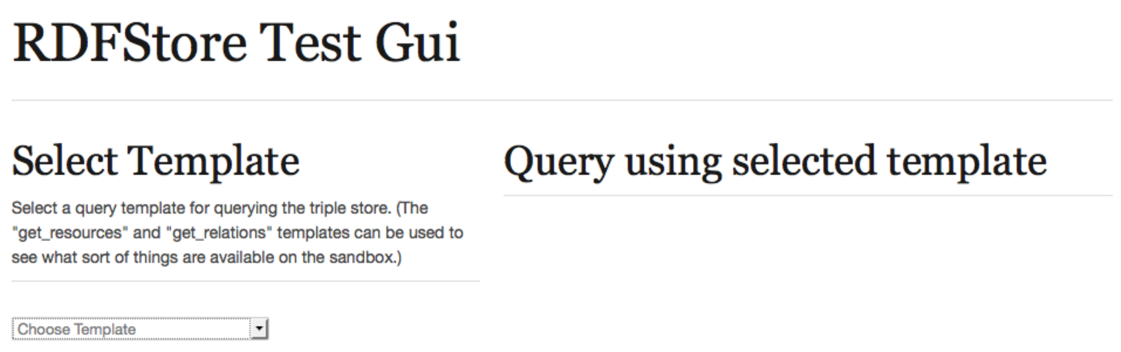
Argument	Description	Example or default
-templates <directory>	Specifies the location of the directory containing Rdfstore templates.	Default: ./templates
-endpoint <URL>	Specifies the SPARQL query endpoint location for the coordinated triple store.	http://localhost:3030:/chic/query?force-accept=text%2Fplain&output=tsv&query=SELECT... http://localhost:3030/chic/query?force-accept=text%2Fplain&output=tsv&query=
-method <GET or POST>	Specifies which HTTP method your SPARQL endpoint uses.	Default: GET
-update <URL>	Specifies a separate address for updates, when applicable.	Default: copies "endpoint"
-updatemethod <GET or POST>	Specifies the HTTP method for a separate address for updates, when applicable.	Default: copies "method"

-format <format>	A string, containing "%s". The %s will be replaced by the query itself, useful for things like triplestore-specific preambles, etc.	Default: %s
-port <number>	Specifies which port Rdfstore will listen for connections on.	Default: 20060
-help	Displays a help screen.	

5.1.4 Simple GUI

While the goal of Rdfstore 2.0 is to deliver an API, it comes with a built-in simple GUI mainly for the purpose of illustration and education. When Rdfstore is running, the GUI can be accessed at http://HOST:PORT/gui/. For example, if HOST is "localhost" and PORT is "20060", the GUI would be accessible within a Web browser at:

http://localhost:20060/gui/



RDFStore Test Gui

Select Template

Select a query template for querying the triple store. (The "get_resources" and "get_relations" templates can be used to see what sort of things are available on the sandbox.)

Choose Template

Query using selected template

Figure 17 Simple Rdfstore 2.0 GUI

5.1.5 Template system

SPAQRL is the query language for RDF data. In our context, a ***template*** is a SPARQL query which can comport up to ten parameters. Rdfstore reduces SPARQL to a matter of filling-in-the-blanks, namely, one blank for each parameter. Templates can be written specifically to answer specific metadata management needs. Furthermore, a given template may therefore be used while varying the values of its parameters.

Example: The SPARQL query to find all things which are "part-of" the class "acids" might look like so:

```
SELECT DISTINCT ?part
WHERE
{
 ?part <http://example.com/ontology#part-of> <http://example.com/ontology#acids>
}
```

Now suppose you want a generic form for "find all things 'part-of' the class 'X'", where the end-user fills in X. Create a template file with a name like "get_parts_of.txt" with contents:

```
SELECT DISTINCT ?part
          WHERE
          {
           ?part <http://example.com/ontology#part-of> <[0]>
          }
```

Here, [0] is a variable. Other available variables are [1] through [9].

Templates should be stored in a template directory in the form of a text file. When you run Rdfstore, use the command line to tell Rdfstore which directory the templates are stored in (unless you use the default directory). The template's name (minus ".txt") will become part of Rdfstore's GUI. Assuming the template in the above example has been loaded by Rdfstore, the template can be accessed at an address like

http://yoururl.org:20060/get_parts_of/?0=acids

Adding template to a running Rdfstore instance is not supported and the addition of templates requires restarting Rdfstore.

5.1.5.1 Advanced Template Commands

At the beginning of a template file, certain special commands can be issued. You can give a name to a variable, as in the following example:

```
# 0 = whole
SELECT DISTINCT ?part
WHERE
{
 ?part <http://example.com/ontology#part-of> <[0]>
}
```

In this example, the command is that first line, # 0 = whole. It says that the name of the variable 0 is 'whole' (so the template is searching for 'parts' of the 'whole'). This is how the Rdfstore demo GUI knows which placeholder text to put in the different form fields.

The other type of command you can use here is a preprocessor command, as in the following example:

```
# 0 = whole
# Preprocessor0 = http://open-physiology.org:20080/terms/%s?longURI=yes&json=yes
SELECT DISTINCT ?part
WHERE
{
 ?part <http://example.com/ontology#part-of> <[0]>
}
```

The command,

# Preprocessor0 = http://open-physiology.org:20080/terms/%s?longURI=yes&json=yes

indicates that the contents of variable 0 will be passed through the indicated preprocessor. For example, if the user enters 'FMA_50801' for variable 0, Rdfstore will replace the '%s' in the Proprocessor0 string with 'FMA_50801' to get the URL:

http://open-physiology.org:20080/terms/FMA_50801?longURI=yes&json=yes

which points to OWLKB and gets a list of subclasses of FMA_50801. Rdfstore will use that list of subclasses, and query the triplestore for all things which are part-of any subclass of FMA_50801.

5.1.5.2 Rdfstore API

Rdfstore has a dynamic API. The API is defined by the templates loaded when Rdfstore is started. For each template, there is a corresponding API command. If the template is named X.txt, and depends on parameters [0], [1], and [2], then the API command looks like:

http://example.com:20060/X/?0=fill_this_in&1=also_fill_this&2=this_too

5.1.6 Low Level services

We refer to services as low level services when they are mere syntactic variations on basic SPAQRL commands. There are three kinds: commands to add a record and commands to delete a record as well as commands to query records.

5.1.6.1 Query

A low level command allows wrapping (URL encoded) SPARQL queries.

For example: select ?x ?y ?z where {?x ?y ?z} limit 10

Can be invoked as:

http://localhost.org:20060/Raw_SPARQL/?0=select%20%3Fx%20%3Fy%20%3Fz%20where%20{%3Fx%20%3Fy%20%3Fz}%20limit%2010

5.1.6.2 Insertion

A low level command allows inserting a triple (SPAQRL INSERT DATA):

http://localhost:20060/Insert_Triple_%28Fuseki%29/?0=a&1=b&2=c

5.1.6.3 Deletion

A low level command allows inserting a triple (SPAQRL INSERT DATA):

http://open-physiology.org:20060/Delete_Triple_%28Fuseki%29/?0=a&1=b&2=c

5.1.7 Specific Triplestores Documentation

Specific documentation for using Rdfstore with individual triplestores: Virtuoso and Fuseki.

5.1.7.1 Virtuoso

When your server is running Virtuoso, by default the SPARQL endpoint is on port 8890. In the following documentation, we'll assume you keep that default; if you change it to another port, then change everything accordingly.

5.1.7.1.1 Queries

Depending on what format you'd like the results in, you can use one of the following strings as the "endpoint" when running Rdfstore.

5.1.7.1.1.1 JSON format

Endpoint string:

http://localhost:8890/sparql?default-graph-uri=&format=application%2Fsparql-results%2Bjson&timeout=0&debug=on&query=

Minimum working example commandline:

```
java Rdfstore -endpoint "http://localhost:8890/sparql?default-graph-
uri=&format=application%2Fsparql-results%2Bjson&timeout=0&debug=on&query="
```

5.1.7.1.1.2 HTML format

Endpoint string:

http://localhost:8890/sparql?default-graph-uri=&format=text%2Fhtml&timeout=0&debug=on&query=

Minimum working example commandline:

```
java Rdfstore -endpoint "http://localhost:8890/sparql?default-graph-
uri=&format=text%2Fhtml&timeout=0&debug=on&query="
```

5.1.7.1.1.3 Other formats

Virtuoso makes a lot of other formats available. To see the list, go to this Virtuoso SPARQL documentation page and scroll down to "16.2.3.3.3. Response Format".

For each listed content type, the general formula for the endpoint string is:

http://localhost:8890/sparql?default-graph-uri=&format=(content type)&timeout=0&debug=on&query=

where (content type) is replaced by the url-encoded mimetype from the above link.

Example:

Suppose you want the format as "application/x-turtle".

Urlescape to get: "application%2Fx-turtle".

The endpoint string is:

http://localhost:8890/sparql?default-graph-uri=&format=application%2Fx-turtle&timeout=0&debug=on&query=

5.1.7.1.2 Adding triples

There are two things to know to set up triple-authoring via Rdfstore via Virtuoso.

5.1.7.1.2.1 Must specify graph

When adding a triple in Virtuoso, it is necessary to specify which graph it goes in. Here's an example "Insert_Triple.txt" template:

```
# 0 = Graph IRI
# 1 = Subject IRI
# 2 = Predicate IRI
# 3 = Object IRI
INSERT INTO <[0]>
{
  <[1]>
  <[2]>
  <[3]>
}
```

Note that the four comments at the beginning are just to tell the GUI what placeholder text to put in the blank fields; they aren't strictly necessary.

5.1.7.1.2.2 Must grant permission

By default, Virtuoso forbids triple-insertion via SPARQL endpoint. If triple-insertion is forbidden, then your triple-insert Rdfstore templates will fail.

Here's how to enable triple-insertion via SPARQL endpoint:

- Connect to Virtuoso's ISQL console. From the command line on the machine where Virtuoso is running, this is usually done with the "isql" command (or "isql-vt" on Ubuntu).
- Issue the command:

```
GRANT execute ON SPARQL_INSERT_DICT_CONTENT TO "SPARQL";
```

- You might be prompted for your Virtuoso credentials; if so, enter them.
- Issue the command:

```
GRANT execute ON SPARQL_INSERT_DICT_CONTENT TO SPARQL_UPDATE;
```

- If you also want to enable templates to delete triples, issues the following commands as well:

```
GRANT execute ON SPARQL_DELETE_DICT_CONTENT TO "SPARQL"
GRANT execute ON SPARQL_DELETE_DICT_CONTENT TO SPARQL_UPDATE;
```

Note: If you are worried about the security implications of allowing triple-insertion via SPARQL endpoint, our recommendation is as follows. You should conFigure your machine so that only localhost is permitted to connect to port 8890 (or whichever port Virtuoso is running on). Then, you can perform proper validation of user input in whatever program it is you're designing, before invoking the Rdfstore API.

5.1.7.2 Fuseki

By default, the Fuseki triple-store runs a SPARQL endpoint on port 3030. If you're running Fuseki on some other port, change everything accordingly.

When using Fuseki, one gives one's dataset a name, and that name has to be inserted into the SPARQL endpoint URL. For the documentation below, we will assume your dataset is named "dataset". If you use a different name, change everything accordingly.

5.1.7.2.1 Queries

Depending on what format you like, you can run Rdfstore with the following endpoint strings.

5.1.7.2.1.1 JSON

Endpoint string:

http://localhost:3030/dataset/query?force-accept=text%2Fplain&output=json&query=

Minimum working example command line (query only, no update support):

```
java Rdfstore -endpoint "http://localhost:3030/dataset/query?force-
accept=text%2Fplain&output=json&query="
```

Remember to change "dataset" to the actual name of your Fuseki dataset!

5.1.7.2.1.2 Text

Endpoint string:

http://localhost:3030/dataset/query?force-accept=text%2Fplain&output=text&query=

Minimum working example command line (query only, no update support):

```
java Rdfstore -endpoint "http://localhost:3030/dataset/query?force-
accept=text%2Fplain&output=text&query="
```

Remember to change "dataset" to the actual name of your Fuseki dataset!

5.1.7.2.1.3 XML

Endpoint string:

http://localhost:3030/dataset/query?force-accept=text%2Fplain&output=xml&query=

Minimum working example command line (query only, no update support):

```
java Rdfstore -endpoint "http://localhost:3030/dataset/query?force-
accept=text%2Fplain&output=xml&query="
```

Remember to change "dataset" to the actual name of your Fuseki dataset!

5.1.7.2.1.4 Tab Separated Values (TSV)

Endpoint string:

http://localhost:3030/dataset/query?force-accept=text%2Fplain&output=tsv&query=

Minimum working example command line (query only, no update support):

```
java Rdfstore -endpoint "http://localhost:3030/dataset/query?force-
accept=text%2Fplain&output=tsv&query="
```

Remember to change "dataset" to the actual name of your Fuseki dataset!

5.1.7.2.2 Adding triples

The Fuseki SPARQL endpoint uses different URLs for SPARQL queries and SPARQL updates. Furthermore, it only accepts SPARQL updates sent with an HTTP POST, it rejects updates sent with HTTP GET.

Fortunately, Rdfstore allows you to specify a separate address/method for updates. Run Rdfstore with command line options

```
-updatemethod POST
```

and

```
-update "http://localhost:3030/dataset/update"
```

(replace "dataset" with the actual name of your dataset).

Minimal working example

If you want to run Rdfstore using Fuseki as the triplestore, returning query results in JSON format, and with a dataset named "models", you can run it as follows:

```
java Rdfstore -endpoint "http://localhost:3030/models/query?force-
accept=text%2Fplain&output=json&query=" -update "http://localhost:3030/models/update" -
updatemethod POST
```

5.2 *Ontology Knowledge Base and Services*

OWLKB 2.0 is the Ricordo semantic reasoning server. It provides an API for querying semantic data which is loaded from an ontology. OWLKB is smart enough to know the semantic meanings of the terms in the ontology and to act accordingly.

As a simple example, suppose that the ontology says widget X was created at factory Y, and that factory Y only creates blue widgets. A query for "show all blue widgets" will show X even if the ontology does not explicitly say that X is blue: the reasoner is smart enough to deduce the blueness of X from the other two facts.

5.2.1 Installation

- Ensure a java runtime and java compiler are installed.
- Use "`git clone`", or any other means, to copy the repository from http://github.org/semitrivial/owlkb
- Within the directory containing the copy of the repository, expand OWLKB's dependencies using "`jar -xf dep.jar`"
- Within the directory containing the copy of the repository, compile OWLKB using "make" (or on Windows: "`javac -g Owlkb.java`")
- Within that directory, run "`java Owlkb.java -help`" to get help on the command line arguments, or see further below.

5.2.2 Loading an ontology

OWLKB loads ontologies in .owl form; we assume the user has an owlfile on their system. When running OWLKB, one should specify the location of the desired .owl file. This is done using the -file command line argument.

For example:

```
java Owlkb -file /home/ontologies/ricordo.owl
```

5.2.2.1 Command line arguments

OWLKB can be run with the following command line arguments.

Argument	Description	Example or default
-file <location of file>	Specifies which ontology file OWLKB will load.	Default: ./templates
-port <port number>	Specifies which ontology file OWLKB will load.	Default: 20080
-reasoner <elk or hermit>	Specifies which reasoner OWLKB will use.	Supported: Elk, HermiT

-namespace <base of iri>	Specifies the namespace to be used for classes created with OWLKB.	
-save <true or false>	Specifies whether or not OWLKB saves new classes to harddrive	Default: true
-help	Displays a help screen.	

5.2.3 Simple GUI

OWLKB comes with a simple GUI. When OWLKB is running, the GUI can be accessed at

http://localhost:20080/gui/

(replace "localhost" with whatever host you're running OWLKB on, and replace "20080" with whatever port your OWLKB is running on, if necessary).

For example, the demonstration instance of OWLKB is running on host open-physiology.org on port 20080, so the GUI is at http://open-physiology.org:20080/gui.

The built-in GUI is mainly just for demonstration purposes. We anticipate OWLKB will mainly be used directly via the API.

5.2.4 KBCaller Java Library

OWLKB is not designed as a library and is not. The reason for this is that OWLKB is rather resource-intensive when loaded with a non-trivial ontology. Thus it makes more sense as a separate process than as a library. Nevertheless, KBCaller is a Java mini-library which abstracts the act of sending API requests to OWLKB over HTTP and can be used in Java-based projects.

5.2.4.1 Constructor

```
public KBCaller( String url )
```

Creates a KBCaller object. Specify the url of an OWLKB instance, including port.

For example, the OWLKB demo instance has the url http://open-physiology.org:20080. If OWLKB is running on the same machine as the Java application you're working on, and if OWLKB is running on its default port (20080), you can use the url http://localhost:20080

5.2.4.2 API Methods

In all cases except for "addlabel", the methods return a list of results as a JSON list, e.g. something like:

['FMA_50801','CHEBI_999','RICORDO_56345634']

If you would prefer the results as an ArrayList<String> and you don't want to add a full JSON parser dependency to your project, we've included a bare-bones JSON-list-parser function in KBCaller:

```
        public ArrayList<String> parse_json( String json ) throws IOException
```

You can compose this with any of the String-returning API methods (except "subhierarchy"), for example:

```
KBCaller kbcaller = new KBCaller( "http://open-physiology.org:20080" );
String subclasses_raw;
List<String> subclasses;
try
{
  String subclasses_raw = kbcaller.subterms( "part-of some FMA_50801" );
  subclasses = kbcaller.parse_json( kbcaller.subterms( "part-of some FMA_50801" ) );
}
catch( Exception e )
{
  e.printStackTrace();
}
```

If you want to parse the "subhierarchy" JSON, you'll probably want to use a full JSON parser for that, as it's not a flat list.

5.2.5 OWLKB API

OWLKB launches a server which listens for connections and responds to the following types of requests.

Note: The "eqterms" type of request is special. Unlike the other commands, "eqterms" will actually create a new class and add it to the selected ontology, if no equivalent class already exists. This is one of the main features of OWLKB, creation of so-called composite terms.

5.2.5.1 subterms

Finds all subterms of the indicated term. For example, "amino acid" is a subterm of "acid".

Example:

http://localhost:20080/subterms/CHEBI_33709

5.2.5.2 parents

Finds all the direct parents (i.e., the direct superclasses) of the indicated term.

Example:

http://localhost:20080/parents/CHEBI_33709

5.2.5.3 children

Finds all the direct children (i.e., the direct subclasses) of the indicated term.

Example:

http://localhost:20080/children/CHEBI_33709

5.2.5.4 siblings

Finds all siblings of the indicated term. A 'sibling' is defined to be an immediate subterm of an immediate superterm of the indicated term.

Example:

http://localhost:20080/siblings/CHEBI_33709

5.2.5.5 subhierarchy

Finds all subterms of the indicated term, and displays them in a hierarchical format (using JSON).

Example:

http://localhost:20080/subhierarchy/CHEBI_33709

5.2.5.6 eqterms

Finds all terms equivalent to the indicated term. For example, the class of all "animal cells" (CL_0000548) capable of some "reproductive process" (GO_0022414) is equivalent to the class of all "germ line stem cells" (CL_0000039).

If there are no equivalent terms, a new class is created, defined to be equivalent to the indicated term. The new class is saved to the ontology (unless saving to hard-drive was disabled by command-line argument).

Example:

http://localhost:20080/eqterms/CL_0000548+and+(capable_of+some+GO_0022414)

5.2.5.7 terms

Finds all terms and all subterms of the indicated term. Note that unlike "eqterms", this API command will not create a new class if no equivalent classes are found.

Example:

http://localhost:20080/terms/CL_0000548+and+(capable_of+some+GO_0022414)

5.2.5.8 instances

Finds all instances of the indicated class. For example, "IN-VITRO-CCTYPE" might be an instance of "TYPE-OF-CLINICAL-CONTEXT". (This is, of course, only for ontologies that include named individuals; otherwise "instances" will always return the empty result set.)

Example:

http://localhost:20080/instances/TYPE-OF-CLINICAL-CONTEXT

5.2.5.9 labels

Finds all labels annotated to the indicated term (specifically, all rdfs:label's). For example, the label "Brain" is annotated to FMA_50801.

Example:

http://localhost:20080/labels/FMA_50801

5.2.5.10 search

Finds all classes in the ontology with the given label (specifically, the given rdfs:label). Note that this is an exact, case-sensitive search--a search for "Brai" or "brain" will not return "Brain" for instance.

Example:

http://localhost:20080/search/Brain

5.2.5.11 addlabel

Adds a label to a class that was created with "eqterms". For syntax, see the example above. To be more precise, the label which is added is an <rdfs:label>. Multiple labels can be added for a single class. This command triggers OWLKB to save changes to the ontology to the hard drive (unless saving has been disabled via command line).

Example:

http://localhost:20080/addlabel/RICORDO12345=volume+of+blood+in+aorta

5.2.6 JSON

There are three ways to coerce data into JSON format:

1. Include an URL paramater `'json'`.

Example:

http://localhost:20080/subterms/FMA_50801?json

2. Include an URL parameter 'verbose'. In addition to changing the command output to json, this also causes the command to send additional information (most importantly, it will send labels along with terms).

Example:

http://localhost:20080/siblings/FMA_50801?verbose

3. Send a request header "Accept: application/json". This has the same effect as method number 1 from above.

Example:

```
curl --header "Accept: application/json" "http://localhost:20080/subterms/CHEBI_33709"
```

5.2.7 Verbose Results

Because of backward-compatibility considerations, the default form of OWLKB results is sparse (including nothing but raw terms in most cases, whereas the user is probably interested in the labels of those terms as well). In order to get labels along with terms, use the 'verbose' URL parameter. Note that this will also coerce the results into JSON format.

Example:

http://localhost:20080/subterms/CHEBI_33709?verbose

5.2.8 Manchester Syntax

The strength of OWLKB is that in all the API commands where a term is expected, a compound term can be indicated using Manchester Syntax. Of course, when passing Manchester Syntax in an URL, it should be urlencoded.

Here are some examples of Manchester Syntax (we've replaced spaces with +'s so these examples can be used in URLs):

- All subclasses of (GO_0000111 intersect GO_0000112):
    - "GO_0000111+and+GO_0000112"
- All things that are GO_0000111 and part-of some GO_0000112:
    - "GO_0000111+and+part-of+some+GO_0000112"
- All things that are (GO_0000111 intersect GO_0000112) and part-of some GO_0000113:
    - "(GO_0000111+and+GO_0000112)+and+part-of+some+GO_0000113"
- All things that are GO_0000111 and part-of some (GO_0000112 intersect GO_0000113):
    - "GO_0000111+and+part-of+some+(GO_0000112+and+GO_0000113)"

5.3 *LOLS Terminology services*

LOLS stands for Local Ontology Lookup Service. Its intended purpose: for a given set of ontologies, let people look up rdfs:labels from IRIs and IRIs from rdfs:labels. LOLS is lean and minimalist, allowing easy deployment on any machine, removing the need to refer to a centralized label lookup service which might be located on the other side of the world.

Technically, LOLS has two components:

i) A converter which turns an OWL file into a LOLS file. Written in Java to use the OWLAPI.

ii) The main engine, which loads a LOLS file and serves API requests in HTTP. Written in C.

5.3.1 Prerequiste for installation:

- java runtime and java compiler
- C compiler (gcc)

5.3.2 Installation instructions (tested on linux and Mac)

- Use "`git clone`", or any other means, to copy the repository from http://github.org/semitrivial/LOLS

Two subdirectories will be created: "converter" and "server"

- In the converter directory: expand dependencies with "`jar -xf dep.jar`"
- In the converter directory: "`make`" (or "`javac -g Convert.java`"). This creates a "Convert.class" java executable for converting OWL files to LOLS files.
- In the server directory: "`make`" (or "`gcc lols.c srv.c trie.c util.c -o lols`"). This creates an executable "`lols`" for running LOLS.

5.3.3 LOLS file preparation

LOLS loads IRIs and rdfs:labels from an N-Triples file, which can be generated from an OWL ontology file by means of a converter written in java.

Navigate to the LOLS converter directory (created in "Installation" above).

Run the following command:

```
java Convert (OWLfile) >(outputfilename)
```

For example, if your OWL file is located at "/home/ontologies/fma.OWL", and if you want the LOLS file to be called "fma.LOLS", then you would run:

Example command to extract an N-Triples file, `fma.nt`, from an OWL file, /home/ontologies/fma.OWL:

```
java Convert /home/ontologies/fma.OWL >fma.nt
```

It might be necessary to manually edit the LOLS file to remove unrelated output from the top of it, which was placed there by the OWL reasoner. (In a future version of LOLS this step will not be necessary.)

5.3.3.1 Multiple OWL files

If you have multiple OWL files and you want a single LOLS file to cover all of them, what you should do is create a shell ontology (see example below) file which imports all the desired ontologies. Then run the converter on the shell ontology.

Example

For example, suppose you want your LOLS file to cover /home/fma.owl, /home/chebi.owl, and /home/go.owl. Then you can create the following shell ontology and run the converter on it:

```
    <?xml version="1.0"?>
          <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:owl="http://www.w3.org/2002/07/owl#">
            <owl:Ontology rdf:about="http://open-physiology.org/shell-ontology">
              <owl:imports rdf:resource="file:/home/ricordo/ontology/fma.owl"/>
              <owl:imports rdf:resource="file:/home/ricordo/ontology/chebi.owl"/>
              <owl:imports rdf:resource="file:/home/ricordo/ontology/go.owl"/>
            </owl:Ontology>
          </rdf:RDF>
```

By modifying the above example in the obvious way, you can write a shell ontology to cover whatever set of ontologies you like. Then run the converter on it to get the desired LOLS file. (Note:

the url "http://open-physiology.org/shell-ontology" in the example is just a placeholder url, anything will work there and it won't effect the resulting LOLS file.)

5.3.4 Running the LOLS server

Once you've created a LOLS file, you can launch the LOLS server by going to the "server" directory (created in "Installation" above) and running:

```
./lols (path to LOLSfile)
```

For example, if you created the LOLSfile "/home/ontologies/mylols.LOLS", then you can run:

```
./lols /home/ontologies/mylols.LOLS
```

By default, LOLS will open an HTTP server on port 5052. (You can change that in srv.c and re-compile, if you prefer another port.) See "API" (below) and "Built-in GUI" (below) for how to actually use that server.

5.3.5 Simple GUI

LOLS comes with a simple built-in GUI. Assuming the LOLS server is running, you can access the GUI at http://(yourdomain):5052/gui

Example

If your domain is "example.com" then you can access the LOLS GUI at

http://example.com:5052/gui

Of course, if you don't have a domain, an IP address or "localhost" can be used instead.

5.3.6 LOLS API

LOLS launches a server which listens for connections and responds to the following types of requests.

In each case, the results are output in JSON format.

5.3.6.1 iri

Finds all rdfs:labels associated to the class with the specified IRI. The IRI can either be specified in full, as in the second example, or else abbreviated as in the first example.

Example (shortform):

http://localhost:5052/iri/FMA_50801

Example (longform):

http://localhost:5052/iri/http%3A%2F%2Fpurl.org%2Fobo%2Fowlapi%2Ffma%23FMA_50801

Note that "http%3A%2F%2Fpurl.org%2Fobo%2Fowlapi%2Ffma%23FMA_50801" is the urlencoded result of "http://purl.org/obo/owlapi/fma#FMA_50801".

5.3.6.2 label

Finds all IRIs of classes with the indicated rdfs:label (case sensitive). The IRIs are given in full.

Example:

http://localhost:5052/label/Brain

5.3.6.3 label-case-insensitive

Finds all IRIs of classes with the indicated rdfs:label (case insensitive). The IRIs are given in full.

Example:

http://localhost:5052/label/brain

5.3.6.4 label-shortiri

Finds all IRIs of classes with the indicated rdfs:label (case sensitive). The IRIs are given in abbreviated form, if possible.

Example:

http://localhost:5052/label-shortiri/Brain

5.3.6.5 label-shortiri-case-insensitive

Finds all IRIs of classes with the indicated rdfs:label (case insensitive). The IRIs are given in abbreviated form, if possible.

Example:

http://localhost:5052/label-shortiri-case-insensitive/brain

6 Examples of hypermodel annotation service calls

Based on the infrastructure and the approach taken here, a number of specialised service calls can be defined to suit specific use cases and requirements. In a manner of illustration we specify a few basic examples. The tools put in place allow for specifying any further needed specific requests.

6.1.1 Get_Hypomodels

Arguments: none

Description: Returns a once column table of URIs for hypomodels.

Method:	GET
URL:	http://<HOST>:<PORT>/Get_Hypomodels/
Body:	[nil]

Example request: Returns all the hypomodels URIs

```
http://localhost:20060/Get_Hypomodels
```

Example response:

```
<pre>
&#60;http://example/update-base/#model1&#62;
&#60;http://example/update-base/#model2&#62;
</pre>
```

Template .txt

```
select distinct ?model where {
?model <http://www.w3.org/2000/01/rdf-schema#type>
<http://www.chic.eu/ontologies/resource#Model-ChicHypomodel>
}
```

6.1.2 Get_HypomodelInputParameter_ByInterpretation_exactMatch

Argument_0: URL encoded URI

Description: Returns a one column table of URIs of input parameters of any model such that the parameter is annotated with the URI provided as argument.

Method:	GET

URL:	http://<HOST>:<PORT>/Get_HypomodelInputParameter_ByInterpretation_exactMatch/?0=<URL-ENCODED-URI>
Body:	[nil]

Example request: Returns all the input parameters whose interpretation is http://www.chic.eu/ontologies/some-domain-ontology#CHIC_0001023 (a fictional concept for the sake of illustration)

```
http://localhost:20060/Get_HypomodelInputParameter_ByInterpretation_exactMatch/?0=http%3A%2F%2Fwww.chic.eu%2Fontologies%2Fsome-domain-ontology%23CHIC_0001023
```

Example response:

```
<pre>
&#60;http://example/update-base/#parameter2a&#62;
&#60;http://example/update-base/#parameter1&#62;
</pre>
```

Template .txt

```
# 0 interpretation
select distinct ?parameter where {
?parameter <http://www.chic.eu/ontologies/some-model-ontology#interpreted-type> <[0]> .
?parameter <http://www.chic.eu/ontologies/some-model-ontology#input-parameter-of> ?model }
```

6.1.3 Get_HypomodelOutputParameter_ByInterpretation_exactMatch

Argument_0: URL encoded URI

Returns a one column table of URIs of output parameters of any model such that the parameter is annotated with the URI provided as argument.

Method:	GET
URL:	http://<HOST>:<PORT>/Get_HypomodelOutputParameter_ByInterpretation_exactMatch/?0=<URL-ENCODED-URI>
Body:	[nil]

Example request: Returns all the output parameters whose interpretation is http://www.chic.eu/ontologies/some-domain-ontology#CHIC_0001023 (a fictional concept for the sake of illustration)

```
http://localhost:20060/Get_HypomodelOutputParameter_ByInterpretation_exactMatch/?0=http%3
A%2F%2Fwww.chic.eu%2Fontologies%2Fsome-domain-ontology%23CHIC_0001023
```

Example response:

```
<pre>
&#60;http://example/update-base/#parameter3a&#62;
</pre>
```

Template .txt

```
# 0 interpretation
select distinct ?parameter where {
?parameter <http://www.chic.eu/ontologies/resource#hasInterpretedType> <[0]> .
?parameter < http://www.chic.eu/ontologies/resource#output-parameter-of> ?model
}
```

6.1.4 Get_Consistent_HypomodelOutputParameter_ByInterpretation_exactMatch

Argument_0: URL encoded URI

Returns a one column table of URIs of output parameters of any model such that the parameter is annotated with the URI provided as argument.

Method:	GET
URL:	http://<HOST>:<PORT>/Get_Consistent_HypomodelOutputParameter_ByInterpretation_exactMatch/?0=<URL-ENCODED-URI>
Body:	[nil]

Example request: Returns all the output parameters of any hypomodel whose interpretation is exactly the same as that of http://example/update-base/#parameter2a (a fictional parameter URI for the sake of illustration)

```
http://localhost:20060/Get_Consistent_HypomodelOutputParameter_ByInterpretation_exactMatch
```

```
/?0=http%3A%2F%2Fexample%2Fupdate-base%2F%23parameter2a
```

Example response:

```
<pre>
&#60;http://example/update-base/#parameter3a&#62;
</pre>
```

Template .txt

```
# 0 parameter
select distinct ?parameter2 where {
<[0]> < http://www.chic.eu/ontologies/resource#interpreted-type> ?i .
?parameter2 < http://www.chic.eu/ontologies/resource#interpreted-type> ?i .
?parameter2 < http://www.chic.eu/ontologies/resource#output-parameter-of> ?model2
}
```

7 Conclusion

In the present report, we described

- the overall semantic infrastructure supporting semantic services,
- the CHIC component requirements for semantic services,
- The core Knowledge Representation underlying the representation of the data records managed through semantic services,
- the Software housing the backend onto which semantic services are applied,
- the invocation of selected basic examples of semantic services.

We did not delve further into the details of the knowledge representation in the present deliverable.

The work evolves through processes of refinement and iteration and we envision evolutions which more specifically will concern:

- Model Repository services
- Clinical Data Repository Services
- GUI configuration services

We will maintain an online and publicly available repository with the following content:

- Software for the semantic infrastructure
- Ontologies and their documentation
- Template use of the ontologies within the software environment in support of required services.

https://github.com/open-physiology/chic

8 References

[1] D6.1 – Cancer hypomodelling and hypermodelling strategies and initial component models

[2] D8.1 – Design of the CHIC repositories

[3] http://www.mkbergman.com/483/advantages-and-myths-of-rdf/

[4] Schulz, M., Krause, F., Le Novere, N., Klipp, E., & Liebermeister, W. (2011). Retrieval, alignment, and clustering of computational models based on semantic annotations. Molecular systems biology, 7(1), 512.

[5] D8.3 – Implementation of the interfaces of the CHIC repositories

Appendix 1 – Abbreviations and acronyms

RDF	Resource Description Framework
URI	Uniform Resource Identifier
ER	Entity Relationship
OWL	Web Ontology Language
API	Application Programming Interface
LOLS	Local Ontology Lookup Service
OWLKB	OWL Knowledge Base
SPARQL	SPARQL Protocol and RDF Query Language
JSON	JavaScript Object Notation
TSV	Tab Separated Value
CHICRO	CHIC Resource Ontology
FMA	Foundational Model of Anatomy
CHEBI	Chemical Entities of Biological Interest
PATO	Phenotypic Quality Ontology
GO	Gene Ontology
UO	Unit Ontology