Implementation of a Referent Tracking System

Manzoor

Center of Excellence in Bioinformatics and Life Sciences
University at Buffalo, NY, USA
701 Ellicott Street
Buffalo, NY 14203
USA
smanzoor@buffalo.edu

Ceusters

Center of Excellence in Bioinformatics and Life Sciences
University at Buffalo, NY, USA
701 Ellicott Street
Buffalo, NY 14203
USA
Ceusters@buffalo.Edu

Rudnicki

Center of Excellence in Bioinformatics and Life Sciences
University at Buffalo, NY, USA
701 Ellicott Street
Buffalo, NY 14203
USA
rjr9@buffalo.edu

Implementation of a Referent Tracking System

ABSTRACT

Traditional database resources and Semantic Web technology face problems when there is a need to keep track of individuals in reality as these individuals undergo changes of various sorts. We describe an application which implements the Referent Tracking paradigm in which each real world entity has its own unique ID. The application is designed to be able to store relationships between tracked instances and also to be extendable to very high orders of magnitude (in principle to accept numbers of entries in the billions). Our approach is based on ontologies grounded in realism, but it can be extended also to information that is captured using the terminologies or concept-based ontologies used in traditional knowledge representation systems. The repository uses RDF as representation format, and it can thus be queried with query languages such as SPARQL, SeRQL and RQL, thereby providing support for reasoning over multiple ontologies.

Keyword: ontology; referent tracking; unqualified realism; Semantic Web; RDF; Electronic Health Record; semantic interoperability

INTRODUCTION

Electronic Health Record (EHR) systems are software systems that manage patient information that typically arises within a single health care institution. Such systems exist in various flavors and can be built up out of several different types of components and rely on different types of standards such as HL7 (Health Level Seven Inc., 2007) or openEHR (Blobel, 2006). One particular component of a modern EHR deals with the access to terminologies such as ICD-9-CM (U.S. Department of Health & Human Services, 2006) or SNOMED-CT (SNOMED International, 2007), to coding and classification systems, and, more recently, also to ontologies such as the Foundational Model of Anatomy (FMA™ University of Washington, 2006). The purpose of using such systems is to avoid the ambiguities and interpretation problems that often arise when

health professionals use local terminologies (or no terminologies at all) to enter statements in an EHR (Rosenbloom, Miller, Johnson, Elkin, & Brown, 2006). Unfortunately, this goal has thus far been only partly achieved. Using terminological systems of the sorts referred to above, in which the terms are given an intended and (so it is claimed) unique meaning, may indeed, if the system is used properly, reduce but not eliminate the risk of misinterpretation by humans. And, certainly, existing EHRs do not contain enough information of the right sort to enable correct interpretation by software agents. As an example, imagine that John consults a physician for a fracture in his left forearm at time t1 and that the fact of John having such a fracture at that time is registered in John's record by putting the SNOMED-CT code 91419009 in the diagnosis field of the chart describing that particular visit (which took place at a certain place, with a particular physician, on a precise date, and so forth, all data which are registered in the context of that fracture). If John, at time t2, suffers from another fracture in his left forearm and the same procedure is used for registering this new fact, then, there is no obvious way to know whether the very same fracture is referred to, or a second one. This is because codes from terminological systems or ontologies do not identify uniquely the entities to which they are assigned in the context of clinical record keeping. They rather describe what generic category the entities to which they are assigned belong to. Referent Tracking (RT) is a paradigm that was introduced in 2005 in the field of EHR systems and that is intended to avoid such ambiguities through the mechanism of assigning globally unique IDs (called IUIs for 'Instance Unique Identifiers') to the entities on the side of the patients that clinicians refer to when writing statements in an EHR (Ceusters & Smith, 2005). These IUIs are thus explicit references to the real world entities (called particulars in the tradition of philosophical ontology) on the side of patients, including their body parts, diseases, therapies, and so forth.

Secondly, many efforts have been and are being made in developing ontologies and structured vocabularies in different domains to make data understandable by machines. Here interoperability is an indispensable requisite as EHR data are characteristically derived from different software systems. In this context various representation languages have been developed for purposes of ontology development, an example being the W3C recommended Ontology Web Language (OWL) (M. K. Smith,

Welty, & McGuinness, 2004). In addition, there are tools such as Protégé (Gennari et al., 2003), SWOOP (Kalyanpur, Parsia, Sirin, Grau, & Hendler, 2006) and OBO-Edit (OBO-Edit Working Group, 2006) which have been used in building ontologies such as the FMA and the Gene Ontology (The Gene Ontology Consortium, 2007). Reasoning with such ontologies can be done with tools such as Pellet (Sirin, Parsia, Grau, Kalyanpur, & Katz., 2006), Racer (Haarslev & Möller, 2001) and FaCT (Tsarkov & Horrocks, 2006). Some of these representation tools allow only class-level representations, while most current reasoners do not support reasoning over instances at all nor in ways that mirror the relationships between the instances in reality. And finally, existing ontology tools typically fail when they are loaded with large amounts of instance data.

An additional problem exists at the level of the integration of terminologies and ontologies in EHR applications because the latter can refer to terms from different systems each of them represented in distinct language formats.

We describe a software system which implements the Referent Tracking paradigm. This Referent Tracking System (RTS) is able to contain large amounts of data pertaining to real-world entities and their relationships in a way that is consistent with the view endorsed by philosophical realism. The RTS is designed to act as a backbone for other applications such as EHRs. It uses RDF as a representation language, can be queried by means of semantic query languages thereby providing support for reasoning over multiple ontologies. The software is developed in Java and is available as a standalone server application accessible through web services as well as a library which allows client applications to embed the RTS. The server is intended to be hosted by a health institute which serves as the hub for other health institutes (clients). It also contains a visualization component for the graphs stored in the Referent Tracking database.

REFERENT TRACKING

Main principles

The purpose of an RTS is, as its name suggests, to keep track of referents which are entities that exist in reality, i.e. in the spatiotemporal world that surrounds us. In the context of an EHR, the referents are in the first place *particulars* such as John, John's

Left forearm, the specific fracture in John's Left forearm, and so forth. These particulars are instances of universals such as *person*, *Left forearm* and *fracture*, which are represented in ontologies. The term *universal* is a philosophical term used to denote what is general in reality. Universals are represented in ontologies by means of *classes*.

Referent tracking requires explicit reference to be made by means of IUIs to the particulars about which users want to provide information. Thus the information that is currently captured in the EHR by means of sentences such as: "this patient has a left forearm fracture", would need to be conveyed by means of descriptions such as "#IUI-4 is located in #IUI-1", together with associated information to the effect that "IUI-1" refers to the patient under scrutiny, and "IUI-4" to a particular fracture in patient #IUI-1 (and not to some similar left forearm fracture from which he suffered earlier). The RTS must correspondingly contain information relating particulars to classes, such as "IUI-4 instance of fracture" (where 'fracture' might be replaced by a unique identifier pointing to the class *fracture* in an ontology). Following the terminology defined in (B. Smith, Kusnierczyk, Schober, & Ceusters, 2006), a configuration of particulars and or universals is called a *portion of reality*.

Of course, EHR systems that endorse the referent tracking paradigm should have mechanisms to capture the information that describes portions of reality in an easy and intuitive way, including mechanisms to translate generic statements into the intended concrete form, a form which may itself be operative primarily behind the scenes, so that the IUIs themselves remain invisible to the human user. This sort of technology is not addressed in this document.

In (Ceusters & Smith, 2006) the conditions for assigning an IUI to a particular are described, as well as the templates according to which some portions of reality are to be represented in an RTS. An additional template for dealing with what in healthcare is known as "negative clinical findings", is introduced in (Ceusters, Elkin, & Smith, 2006). Finally, at this time, also a template for registering names by which a particular is referred to in reality has been proposed (e.g. "John" as first name for that particular John), but this is something that will be changed in the future. The current set of templates is shown in Table 1. The templates are to be interpreted as an abstract syntax; it

is left to the developers of an RTS to implement the specifications in the most optimal way given the constraints of the environment in which the system has to operate.

Table 1: Abstract syntax and semantics of information templates in a referent tracking system

Template Name	Abstract Syntax	RDFS class
Description		
A-template	$A_i = < IUI_p, IUI_a, t_{ap}>$	*ParticularRepresentation*
Captures the assignment of an IUI_p to a particular at time t_{ap} by the particular referred to by author IUI_a.		
PtoP – template	$R_i = <IUI_a, t_a, r, o, P, t_r>$	*PtoP*
The particular referred to by author IUI_a asserts at time t_a that the relationship r from ontology o obtains between the particulars referred to in the set of IUIs P at time t_r.		
PtoU-template	$U_i = <IUI_a, t_a, inst, o, IUI_p, u, t_r>$	*PtoU*
The particular referred to by author IUI_a asserts at time t_a that the particular referred to by IUI_p instantiate *inst* relation from ontology o with the universal u at time t_r.		
PtoCo-template	$Co_i = <IUI_a, t_a, cbs, IUI_p, co, t_r>$	*PtoCo*
The particular referred to by author IUI_a asserts at time t_a that the particular referred to by IUI_p is annotated by concept code *co* from terminology system *cbs* at time t_r.		
PtoU¯template	$U^-_i = <IUI_a, t_a, r, o, IUI_p, u, t_r>$	*PtoLackU*
The particular referred to by author IUI_a asserts at time t_a that the relation r of ontology o does not obtain at time t_r between the particular referred to by IUI_p and any of the instances of the class u at time t_r		
PtoN-template	$N_i =< IUI_a, t_a, nt_j, n_i, IUI_p, t_r>$	*PtoN*
The particular referred to by author IUI_a asserts at time t_a that n_i is the name of the nametype nt_j assigned to the particular referred to by IUI_p at t_r.		
Meta-template	$D_i = <IUI_d, X_i, t_d>$	
Publication of a description of a portion of reality in the RTS where IUI_d is the IUI of the entity registering X_i in the system, X_i is the information-unit in question (in the form of any other template above), and t_d is a reference to the time the registration was carried out.		

Requirements

Although an RTS can be used independently in a single setting, for instance within a single general practitioner's surgery or within the context of a hospital, the paradigm's real benefits will primarily emerge when it is used in a distributed, collaborative environment, for instance if an RTS is used as a central server to which many health institutes are connected. One and the same patient is often cared for by a variety of healthcare providers, many of them working in different settings, and each of these settings may use its own separate information system. These systems contain different data, but these data often provide information about the same particulars. Under the current state of affairs, it is very hard, if not impossible, to query these data in such a way that, for a given particular, all information available can be retrieved. With the right sort of distributed RTS, such retrieval becomes in very many cases a trivial matter.

Therefore, an RTS should be in line with the following design principles:

- be able to run as a backbone for any EHR system whereby both EHR and RT systems should run independently;
- be able to run on any platform (Windows, Unix, Linux),
- independent of the programming environment in which it has been developed,
- be able to work with multiple health institutes as a single backbone;
- have reasoning capabilities;
- be able to run in a secure box such that only authorized users can access the services of the RT system;
- be able to handle billions of records in a fast and efficient way.

APPLIED TECHNOLOGIES

Object-Oriented Programming and Java

To satisfy the platform independence requirement, we implemented the RTS in the object-oriented programming language Java. In defining the Java-classes (we explicitly use the term 'Java-class' to differentiate such Java-classes from classes that are part of an ontology) and the Java-objects that would be created during their execution, we maintained as far as possible the same principles as dictated by BFO. We took maximal advantage of the Java interface paradigm to design methods without fixed implementation. We also declared many Java-classes to be abstract such that they don't need to supply specific implementations of each method that they contain. This is useful for providing implementations that are general enough to apply to most anticipated extensions of such a Java-class.

Resource Description Framework

In a statement such as "*John (#IUI-1) has a fracture (#IUI-4) in his left forearm (#IUI-3)*" the IUIs form the nodes in a graph whereas the relations between the particulars denoted by the IUIs such as #IUI-3 *part_of* #IUI-1 and #IUI-4 *depends_on* #IUI-3, form the edges in the graph. Therefore, the Resource Description Framework (RDF) (Manola & Miller, 2004) can be used as a representation language.

RDF is based on the idea that the entities (also called *resources*) being described have properties which have values. The data model of RDF contains the following four components.

- **Resources**: a resource is anything which can be named with a Universal Resource Identifier (URI), e.g. the URI for the particular #IUI-3 would be '*http://org.buffalo.edu/RTS#IUI-3*', where the part *http://org.buffalo.edu/RTS#* is the namespace URI and the part IUI-3 is the local name of the particular. In our RDF representations the RT templates are implemented as resources themselves: each RT template resource is therefore prefixed with the RTS name space URI, i.e http://org.buffalo.edu/RTS#. We are using the label prefix *rts:* for the RTS namespace so that the RT template resource rts:IUI-1 is equivalent to http://org.buffalo.edu/RTS#IUI-1.
- **Literals**: atomic values, such as integers, strings and dates.
- **Properties**: a *property* is a specific aspect, attribute or relation of a resource. A property is itself a resource.
- **Statements**: any assertion in RDF is made by a statement. A *statement* is an ordered triple of the form (*subject*, *predicate*, *object*), where subject is a resource, predicate is a property, and object is a literal or a resource.

The RDF framework provides a simple and elegant way for *describing* properties for resources. However, it does not provide any mechanism to *declare* properties for resources. Therefore, RFDS, an extension of RDF, has been proposed by W3C for declaring RDFS-classes (again, we explicitly use the term 'RDFS-class' to distinguish such classes from JAVA-classes and classes in ontologies which are the representations of universals) and their properties and relations. We have mapped the RT templates definitions to RDFS classes as shown in Figure 1; the arrows in the figure represent the direction of the relations.

We have kept the names of the RDFS-classes and properties identical to the RT template names with minor changes. The *A* and *PtoU*$^{-}$ templates are renamed *ParticularRepresentation* and *PtoLackU* respectively. The *IUIp* property of the *A* template is implemented using an *RDF :ID* property. The *P* property in the *PtoP* template, i.e. an ordered list of the particulars, is implemented with an RDFS-class named

PList which is a subclass of the RDF:Seq class (represents an ordered list of resources). We have defined the RT URI rts:type//terminologysystemid/termid to either *universals (u), concepts (co)* or *relations (r)*. The RT URI starts with the *rts:* prefix and the *type* part represents whether this URI represents a universal, a concept, or a relation. For example the URI *rts:u//FMA/Left+forearm* denotes the FMA representation for the universal *Left forearm* and the URI *rts:r//FMA/part* the FMA's *part* relation. We have omitted the properties *o* (ontology id) and *cbs* (coding system id) from the *PtoU, PtoP* and *PtoCo* templates as they are merged in the RT URI for the respective properties, i.e. *u, r* and *co*.

We have introduced an *Identifier* template which is used in situations when for a particular referred to in a patient encounter chart it cannot be determined at that time whether an IUI already has been assigned to it. The Identifier template assigns a unique number to the candidate particular which is ***not*** an IUI because it is not known at that time whether the ID satisfies the requirement of singularity. Because these identifiers are clearly distinguished from IUIs, it is possible to supply the missing information later and to replace the identifier accordingly with an appropriate IUI.

Based on the RDFS schema as shown in Figure 1, each assertion of the RT template in RDF will receive the RDF ID as shown in Figure 2 which contains the graphical representation of the following RT templates:

A	<IUI-1, IUI-10, 12/01/2006.>.
PtoN	<IUI-10, 12/01/2006, Name, John, IUI-1, 12/01/2006>
A	<IUI-3, IUI-10, 12/01/2006.>.
PtoU	<IUI-10, 12/01/2006, instance_of, IUI-3, FMA, Left forearm, 12/01/2006>
PtoP	<IUI-10, 12/01/2006, has_part, OBO_REL , <IUI-1, IUI-3> 12/01/2006>

Figure 2 shows that particular *rts:IUI-1* (*ParticularRepresentation* assertion) enjoys the *has_part* relation (taken from the OBO_REL ontology (B. Smith et al., 2005)) with particular *rts:IUI-3* which is an instance of the FMA *Left forearm* universal. The *rts:pton_1* resource represents the *PtoN* template assertion that assigns the name *John* to the particular referred to by *rts:IUI-1*. The **rts:iuip** property of *rts:ptop_5* resource id (*PtoP* template assertion) tells us that the particular *rts:IUI-1* is the subject of the relation *has_part* whereas the particular *rts:IUI-3*, as indicated by convention by the property *rts:p* is the object; thus *rts:IUI-1* (John) has_part *rts:IUI-3* (instance of the Left forearm).

Figure 1:RT templates RDFS schema diagram.

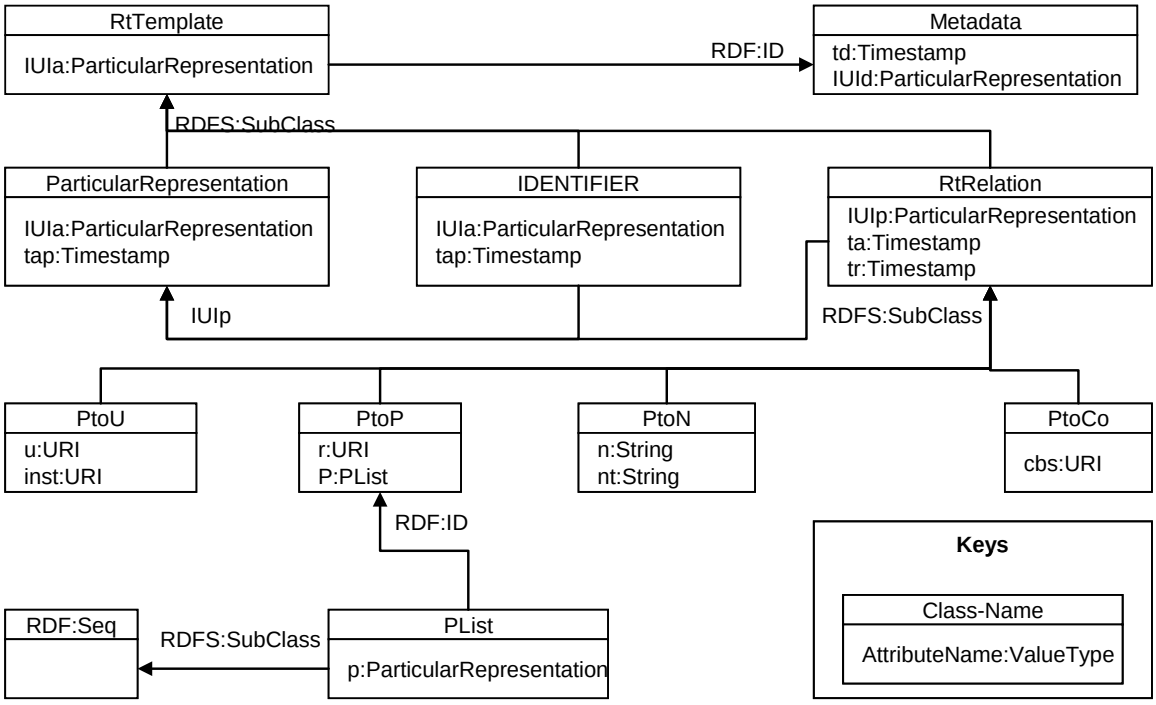
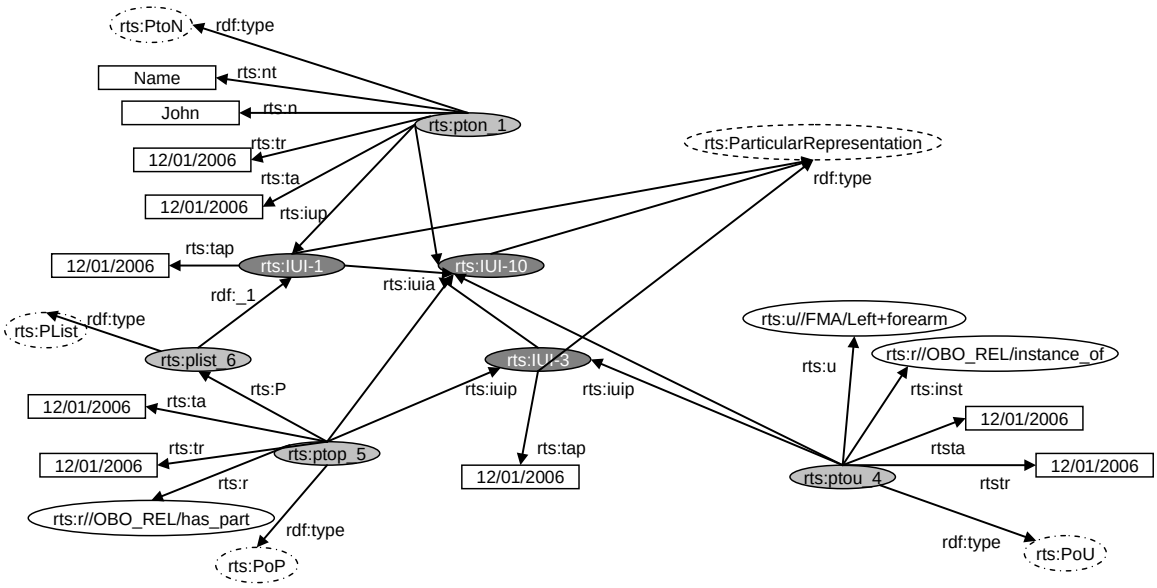


Figure 2: RDF representation for the particular John has has_part relation with his Left forearm

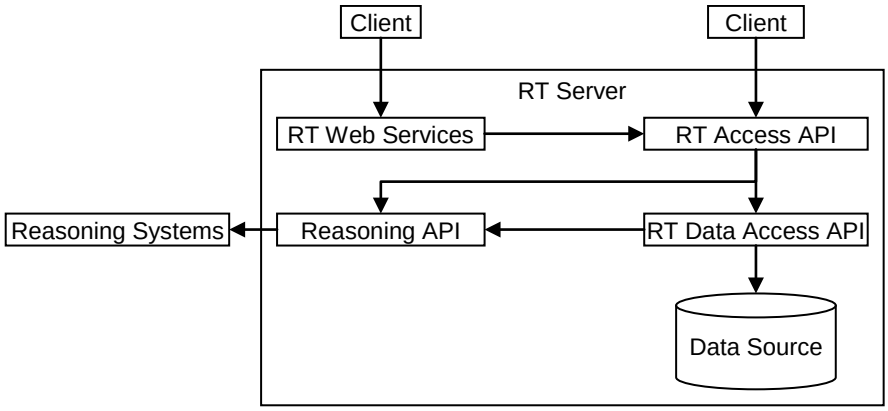


RTS ARCHITECTURE

We have designed the RTS such that it can be used as a server application as well as a Java library. As a server, the system runs as a standalone application inside an apache tomcat HTTP web server at port 8080 (The Apache Software Foundation, 2006) and it can communicate simultaneously with multiple EHR clients running at remote locations. The server is intended to be hosted by a health institute which serves as the hub for other health institutes (clients). The hosting health institute is responsible for taking care of the administration and privacy issues of the shared information stored at the server. The implementation of the RTS is downloadable from Sourceforge under an open source license (Manzoor & Ceusters, 2007).

The architecture of the application is shown in Figure 3. Clients can connect with the system via a server interface based on web services or via the *RT Access API* interface, which is the kernel of the system. The web services forward the clients' requests to the *RT Access API* which is responsible for data validation and management. All data serialization and retrieval activities are performed in the *Data Access Layer*. The reasoning component sends all the reasoning queries (requested by *Data Access API* or *RT Access API*) to external ontology or terminology systems for query execution.

Figure 3: RTS Architecture



RTS Web Services

The web services are remote procedures hosted at an http server which are invoked through messages in line with the Simple Object Access Protocol (SOAP)

(Mitra, 2003). Such messages contain the procedure information (procedure name, parameters and return type) and port type (location of the procedure). SOAP is platform independent so that the RT interface becomes accessible to all programming platforms and environments.

The RT web services interface is written in the Web Services Definition Language (WSDL) (Christensen, Curbera, Meredith, & Weerawarana, 2001), an XML based service description language containing the abstract definition of a service such as service location, message format and protocol bindings (e.g. which protocol needs to be used for communication). The WSDL is also platform independent and clients can call the web services by reading the services details in the WSDL file. RTS uses Axis for Java (The Apache Software Foundation, 2005) to host the web services thereby taking advantage of the native support of WSDL that Axis provides.

The RTS web services allow both retrieval and insertion of the RT templates in the RTS. They run over the http protocol (Booth et al., 2004) which is stateless in nature: both client and server forget each other after processing a request. However, in the RTS, it is required that the server remembers the clients since authentication and other safety and security principles require users to remain logged-in until no data have anymore to be entered or retrieved. To achieve this behaviour, we have used the *session oriented communication paradigm* (Kristol & Montulli, 1997). A session represents a logical connection of a client with the server and is created by the server upon the successful login of the client. The session is expired only if the client logs out from the system or if a timeout occurs. A session is uniquely identified by means of a unique session ID which is generated when the session is created. The client uses this session ID for any further communication in the context of the session.

RT Access API

All the functionalities that the RTS is able to provide to clients are implemented in the API module. The *Webservices* component forwards all requests to this API for execution. As an alternative, Java clients can embed the RTS in their applications using this API. This API contains the modules RTRepository and RTVisGraph.

RTRepository

The RTS has been built to be independent of any data source technology. To achieve this goal, we have defined *RTRepository* as an abstract Java-class. This Java-class provides all necessary services for managing the data based on the principles defined in the RT paradigm. It implements by default all services which are independent of any data source technology while the remaining services are defined as abstract. To manage the RT data in a specific data source technology, an extension of the *RTRepository* for that specific technology is required. We have decided to develop the *RTRepositorySesameImp* Java-class by extending the *RTRepository* such that it targets the SAIL API as a data source. The SAIL API is a Sesame API for manipulating RDF graphs (Broekstra, Kampman, & Harmelen, 2002).

RTRepositorySesameImp works with the three data sources supported by Sesame: one contained in a RDBMS, another one in memory, and the third one being file-based. The RDBMS data source allows maintaining a large repository in the central server. The memory based repository is designed to maintain RT data for temporary purposes. For example, if a client receives RT templates in response of a service execution, the memory based repository can be used to iterate quickly over the results.

At runtime an *RTRepository* instance is created by an instance of the *RTRepositoryFactory* class. The factory class construction helps in creating an instance of a specific implementation *RTRepository* without the need to change the RTS java code. It gets the repository implementation class information (currently *RTRepositorySesameImp*) from the RTS configuration file which maintains the different configurations of the RTS such as initialization parameters. This allows other implementations of the *RTRepository* to be plugged in.

The *RTRepository* services use java objects in their arguments and returned results, and the java objects carry the information about the RT templates. The objects are the instantiation of the java classes which are mappings of the RDFS classes.

The *RTRepository* class provides three types of services, i.e. insertion, retrieval and querying (with semantic query languages) of the RT data.

Insertion services allow creating a new RT template in the repository. The most basic service assigns an IUI to a real world entity and creates its representation in the

RTS. The java code '*ParticularPresentation particular = repository.createParticularRepresentation(tap, IUIa);*' creates first a *ParticularRepresentation* template in the repository, assigns the metadata properties *IUId* and *td*, and then returns the instance.

The next step is to assign detail to this particular. For example, the code '*PtoU ptou = repository.createPtoU(particular.getIUI(),IUIa, "rts:r//OBO_REL/instance_of", "rts:u//FMA/Left+forearm", ta, tr);*' relates the particular created earlier to the *Left forearm* class (represented with *PtoU* template) of the FMA by means of the *instance_of* relation from the OBO relation ontology.

Importantly, the RT paradigm does not allow any deletion operation in order to be able to always return to a state of the database as it was at a certain time in history. To avoid mistakes in creating new templates in the *RTRepository,* the templates are cached right after the create operation. The client can remove or modify the templates from the cache as long as the commit service has not been called.

The API retrieval methods help in searching the particulars in the RT repository. Particulars can be searched by means of the names associated with them, the ontology classes of which they are instances, or the creation and observation dates (*Table 2*).

Table 2: The RTRepository retrieval services to search particulars by means of their associated detail

Service Name	Service Description
getParticularsWithPtoN (*iuip, nt, n, iuia, taRange, tdRange*)	This service retrieves the particulars and the associated *PtoN* templates. The query '*getParticularsWithPtoN (null, "name", "John", null, null, null)'* (which particulars have the name *John)* will for the data shown in Figure 2 retrieve the templates with resources ids *rts:pton_1* and *rts:IUI-1*.
getParticularsWithPtoCo (*iuip, co, iuia, taRange, tdRange*)	This service retrieves the particulars and the associated *PtoCo* templates. The query '*getParticularsWithPtoCo (null, "rts:co//SNOMED-CT/91419009", null, null, null)'* retrieves the particulars annotated with the SNOMED-CT code '91419009', which is a code for *Left forearm fracture.*
getParticularsWithPtoU (*iuip, u, iuia, taRange, tdRange*)	This service retrieves the particulars which are instances of the universal *u*. The query '*getParticularsWithPtoU (null, "rts:u//FMA/Forearm", null, null, null)*' retrieves the instances of the FMA class denoting the universal *Forearm.*
getParticularsWithPtoLackU (*iuip, u, iuia, taRange, tdRange*)	This service retrieves the particulars which do not stand in any *lacks* relation to the universal *u*.

All arguments in the above services can be null. Because the search pattern in the services might match with several thousands of particulars and the network bandwidth might not allow the transfer of that many results to the clients, we have set the limit by default to return the first 200 templates. What selection will be returned depends on the data source technology. However, the limit can be changed in the RTS configuration file.

In *RTRepository*, particulars are connected to each other via relations such as the *has_part* relation between *John* (#IUI-1) and his *Left forearm* (#IUI-3) as shown in Figure 2. We have exploited these relations for retrieval as well and designed services to search particulars by means of the relations through which they are connected. Some examples are shown in *Table 3*.

Table 3: The RTRepository retrieval services to search particulars connected to each other by ontologies relations.

Service Name	Service Description
getParticularsWithPtoPByPtoU (iuip, r, u, r2)	This service retrieves all the particulars that stand in relation *r* to particular *iuip* and which are instances of a universal *u1* which according to the ontology in which *u* and *r2* are represented, stands in the *r2* relation to *u*. When the query *'getParticularsWithPtoPByPtoU (rts:IUI-1, rts:r//OBO_REL/has_part, rts:u//FMA/Upper+limb, rts:r//FMA/part+of)'* is executed over the data in Figure 2, it returns the particular rts:IUI-1 (John), his left forearm (represented by rts:IUI-3) and the relation *has_part* (represented rts:ptop_5) assuming that in the FMA *Left forearm* has the *part of* relation with *Upper limb*.
getParticularsWithPtoPByPtoN (iuip, r, nt, n)	This service retrieves all the particulars with the name *n* of nametype *nt* and that stand in relation *r* to particular *iuip*. The query '*getParticularsWithPtoPByPtoN (rts:IUI-1, null, name, Manzoor*)' would retrieve the particulars whose lastname is Manzoor and that stand in some relation to John (IUI-1).
getParticularsWithPtoPByPtoCo (iuip, r, co)	This service retrieves all the particulars which are annotated with *co* and that stand in relation *r* to particular *iuip*. The query, for example, *'getParticularsWithPtoPByPtoCo(rts:IUI-3,null rts:co//SNOMED/91419009)*' would return the fractures (annotated with SNOMED code 91419009) that have occurred on John's Left forearm.

Querying the RTS using SPARQL

Because the RT data are expressed in RDF, RDF query languages such as RQL (Foundation for Research and Technology – Hellas, 2003), SPARQL (Prud'hommeaux & Seaborne, 2006) and SeRQL (Broekstra et al., 2002) can be used for retrieval. To this end, the *RTRepository* comes with the service '*repository.query(querystring, language)*' which has an argument for the query string and a second one for the name of the query

language in which the first argument is expressed. The SeRQL query language is implemented with the help of the Sesame SeRQL query language module, and the SPARQL query language is implemented with the help of the ARQ query module (a SPARQL processor for Jena) (RDF Data Access Working Group, 2007). Because the RTS repository is built over the Sesame *RDFRepository*, the interoperability between the Sesame *RDFRepository* and Jena is done by means of a modified version of the Jena Sesame Module. The RQL query language is supported by Sesame SAIL but this has thus far not been tested within the context of the RTS. We will limit our discussion here to SPARQL.

Listing 1 enumerates the triples involved in the graph shown in Figure 2 representing that #IUI-3 (John's left forearm) is part of #IUI-1 (John).

Listing 1: Triple View of the RDF

```
1  rts:IUI-1    rdf:type  rts:Particular
2  rts:IUI-1    rdf:tap   "12/01/2006"
3  rts:IUI-3    rdf:type  rts:Particular
4  rts:IUI-3    rdf:tap   "12/01/2006"
5  rts:ptou_4   rdf:type  rts:PtoU
6  rts:ptou_4   rts:iuip  rts:IUI-3
7  rts:ptou_4   rts:u     rts:t//FMA/Left+forearm
8  rts:ptou_4   rts:ta    "12/01/2006"
9  rts:ptou_4   rts:tr    "12/01/2006"
```

SPARQL works with query triples that look very similar to RDF triples, but that may contain variables instead of constants for subject, predicate or object. For example, the query '*SELECT ?r WHERE{?r* ***rts:iuip*** *<rts:IUI-3>.}*' has two clauses: SELECT and WHERE. The SELECT clause contains the variable declaration, and the WHERE clause contains the query search patterns. Because the variable is placed in the subject position, the query returns the list of subjects from matching triples. The pattern at line 3 of the query (no restriction for the subject) matches the one triple at line 6 in *Listing 1*. The returned result is "rts:ptou_4", i.e. the URI of the matching RT template. The query '*SELECT ?ptou 2 WHERE{?ptou* ***rts:u*** *rts:t//FMA/Left+forearm.}*' requests the resources (*PtoU* templates) which are related to the universal *Left forearm*.

The *RTRepository* executes the queries, whether in SPARQL or another supported query language, in two steps. In the first step, it passes the query to the corresponding query engine which upon execution returns the results. The results are the URI for the RT

templates. The repository, in the second step, then queries the Data Access API (described further down) to retrieve all the attributes of the returned templates URI.

Of course, users of EHR systems are not expected to query the *RTRepository* directly through RDF query languages. Rather, these queries should be generated on the basis of the graphical user interfaces provided by the EHR systems. Making that happen is part of the work to be conducted when interfacing an EHR with the RTS.

RTVisGraph

This component is an extension of the JGraph java library for displaying graphs (JGraph Ltd, 2006) which has the ability to generate images in Jpeg. The component can be used for interactive query expansion using the query services just described. A search for *any fracture on John's forearm* for instance can be executed in three steps. In the first step, John (rts:IUI-1) is searched. In the second step, the graph is expanded for the related particulars, in this case rts:IUI-3 (*Left forearm*). Finally the graph expands further from rts:IUI-3 by retrieving the related particulars which are annotated by concept codes, in this case rts:IUI-4 (annotated by SNOMED fracture code).

RT Data Access API

This is the low level data access API which provides persistence services for the RT repository. It provides an abstract view of the data source to *RTRepositorySesameImp*. This layer utilizes currently the services of SAIL to store and retrieve RDF graphs, while the Jena API (HP Labs Semantic Web Research, 2006) for RDF manipulation which does the same job as SAIL might be another choice. SAIL comes with an *RDFRepository* java interface, which represents a logical data repository for RDF graphs and comes with three services:

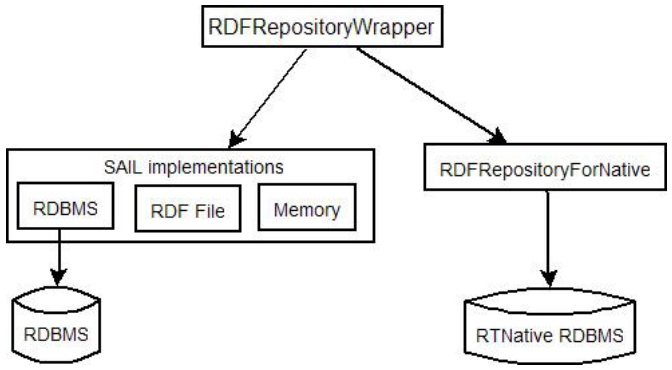
- **addStatement (subject, predicate, object):** This service inserts a new RDF statement in the repository.
- **getStatements (subject, predicate, object):** This service takes a search pattern as a 3-place argument and returns the RDF statements matched with the pattern. Any argument in the pattern could be null.

- **hasStatement (subject, predicate, object):** This service checks whether a triple pattern exists in the repository or not. Its argument pattern is similar to the *getStatements* service.

The *RTRepositorySesameImp* class calls these basic services for all kinds of retrieval and insertion operations. For example, at some point during a query execution, the *RTRepositorySesameImp* class creates a *PtoU* resource as '*PtoU ptou = new PtoUImp("ptou_4")*'. In this example the node object is created by passing the id of the *PtoU* template. *PtoUImp* is the implementation of the *PtoU* interface for the SAIL API. For retrieving the universal *u* and the other properties, the object calls the *getStatements* service. In a similar way to insert a RT template in the repository the *RTRepositorySesameImp* class calls the *addStatement* for each attribute (such as *ta*, *tr*, *iuid* an *iuia* etc) of a RT template.

As a further improvement, rather than using the implementations of the SAIL API directly, we have written the *RDFRepositoryWrapper* over the implementations of SAIL. The purpose of the wrapper is to call under certain circumstances the reasoning services described below.

Figure 4: RDFRepository Implementations



Finally, because the Sesame default implementation for RDBMS is efficient during retrieval, but slow during insertions in large repositories, we have implemented an *RDFRepository* interface for the RTS native database so called *RDFRepositoryForNative* which is more efficient than the Sesame default implementations for both retrieval and insertion.

Reasoning API

Reasoning is a core part of the RTS and its purpose is double: first to avoid inconsistent data from being entered, and second to draw inferences during the execution of the search queries using the generic knowledge expressed in the ontologies used to annotate the data and by exploiting the reasoners that operate on them. Various reasoners exist, some being specific to a particular ontology such as the OQAFMA reasoner of the FMA (Mork, Brinkley, & Rosseb, 2003), some coming with a DIG interface (Bechhofer, 2003) for description logic representations while others use directly OWL-DL.

In order to be able to deal with ontologies of various sorts and their associated reasoners, we developed the Reasoning API which helps in sending reasoning queries uniformly to different ontology systems. The API has an abstract class called *OntologyConnector* which provides an interface to the external ontology systems. The *OntologyConnector* interface services (as shown in *Table 4*) are designed based on the principles defined in the OBO Relation Ontology (B. Smith et al., 2005) and Basic Formal Ontology (Grenon, Smith, & Goldberg, 2004). The interpretations of the *OntologyConnector* services are specific to a particular ontology system; therefore, a separate implementation of the *OntologyConnector* is required for each ontology which is used to annotate the particulars in the RTS. Currently, we have only implemented an extension of the *OntologyConnector* for the FMA, because this is the only one thus far that has a broad coverage and is built on sound ontological principles. Later we will add implementations for OWL based ontologies.

Table 4: OntologyConnector class services

Service Name	Service Description
isUniversalExist(u):	This service checks whether a universal *u* exists in the ontology system.
isRelationExistBetweenUniversals (u1, r, u2):	This service checks whether the relation (r) exists between universals *u1* and *u2*, e.g. *Left eye subclass Eye* and *Left forearm partOf Upper limb*.
getRelations(u1, u2)	This service returns the list of the relations that exist between two univerals *u1* and *u2*.

Description logics are widely used for building ontologies. The reasoners for such ontologies may take from 1 second to a day to compute inferences over the ontology classes depending on their size and definitional complexity. Therefore, instead of directly

communicating with the reasoners for each ontology, it is better to compute all inferences at one time and then store the inference graph in a database; in OWL-DL ontologies the inference graph can be stored in Jena repositories easily as they have native support of OWL.

The execution time of the *OntologyConnector* services can range from milliseconds to minutes, depending on the query execution time in the external ontology system. To handle this issue, the *OntologyConnector* caches the results returned from these systems. The cache is stored in a RDBMS. During the execution of any of the *OntologyConnector* services, it first searches in the cache.

Reasoning is performed for any query which involves *PtoU* templates. If, for example, the query *getParticularsWithPtoPByPtoU(rts:IUI-1,rts:r//OBO_REL/has_part,rts:u//FMA/Upper+limb, rts:r//FMA/part+of)* is executed over the data of *Figure 2*, then first all particulars which are related to the particular rts:IUI-1 via the *has_part* relation are retrieved; in this case rts:IUI-3. Then it retrieves the universals which annotate *rts:IUI-3* by retrieving the *rts:potu_4*. Finally, it requests the ontologies in which the universals are represented by calling the *isRelationExistsBetweenUniversals("Left forearm", "Upper limb", "part of")* service from the *OntologyConnector* instance of the specialized class implemented for the FMA ontology. If the service returns true, then it returns the resulting particulars with their associated templates.

RESULTS

To check the performance and stability of the RTS, we have tested the system by running the search queries over various database sizes up to 1.3 million RT templates. To that end, we developed a *DataGenerator* module which generates data on the basis of two sorts of XML files. Files of one sort contain lists of patients' names. Files of the other sort (term list file) contain lists of body part universals from the FMA ontology including an ICD9 code to indicate a possible pathology associated with that body part. The *DataGenerator* tool first generates the patient particulars by parsing the patient names list and then for each patient it generates randomly a number of body parts particulars and disease particulars associated with the body parts by parsing the term list

configuration file. The result is that for each patient a random number of body parts were declared to be instances of universals that are represented in the FMA and associated with the respective patients by means of the *has-part* relation as defined in the OBO-Relation ontology (B. Smith et al., 2005). To each body part, we associated a disease via the *depends-on* relation. The disease particulars are annotated with ICD9 codes.

To test the retrieval capabilities of the RTS, we randomly picked three patients from the database. The first was related to 22 particulars, the second to 46 and the third with 74. The test case contained 16 queries which ran over the three patients. Each query involved a combination of several services. During the test, inferences about the FMA universals were taken from the cache to exclude in the analysis any computation time for which the RTS is not responsible. All tests were run on the same machine with an Intel Core 2 duo E6400 processor, Windows XP as operating system, 1 GB RAM and the My SQL database 5.1.

Table 5 compares the retrieval times in milliseconds for *RDFRepositoryRTNativeImp* (RT native RDBMS RDF repository) and *RDFRepository* (Sesame RDF repository for RDBMS) obtained by averaging the results of 16 tests. The retrieval time increases as the database size increases, but not at the same rate.

Table 5 Comparison between the RTS native and Sesame RDBMS persistence for the query execution performance by evaluating the query set different data sizes of the RTRepository.

9	# of RT Templates	10	# of Particulars	Query set execution time in milliseconds. 11 12	in the RTS native persistence	13 14	in the Sesame native persistence
15	162552	16	51706	17	195	18	214
19	350075	20	111300	21	200	22	230
23	540430	24	171818	25	214	26	237
27	788143	28	250663	29	219	30	250
31	1279908	32	406360	33	477	34	600

RELATED WORK

The idea of representing particulars in computer systems is not new. (Borgida & Brachman, 1993) reports on a DL system which represents knowledge as consisting of individuals and their relations, where a set of individuals having a similar behavior are represented as a member of a concept. Such systems are built to compute inferences over the concepts (TBox) and individuals (ABox), e.g. individual John is an instance of the patient concept and the patient concept is sub-type of the person concept. However, as most DL systems are built on the concept based paradigm exclusively, where a concept may not correspond to reality (Ceusters, Smith, & Flanagan, 2003), a consequence is that even the most powerful DL may lead to conclusions that might not correspond to reality. Instance store is an example of another system which is built along the same lines of DL systems. Based on OWL-DL, it maintains a large pool of instances (Bechhofer, Horrocks, & Turi, 2005), but excludes relations between them, hence the name '*role free ABox*'.

The Digital Object Identifier (DOI) system has been developed to keep track of the identification, trading, protection, and monitoring of all forms of rights over both tangible and intangible assets. The system allows to assign a unique string to an entity but exhibits some problems at the level of the definitions used to describe what type of instance is dealt with. In (Ceusters & Smith, 2007) the RT paradigm is proposed as solution.

In (Bouquet, Stoermer, Mancioppi, & Giacomuzzi, 2006) the OKKAM system is proposed to keep track of the identifiers and names assigned to entities, and this in response to the inadequacy of URIs to unambiguously identify entities. OKKAM, however, does not deal with relations between the entities.

CONCLUSION & FUTURE WORK

In this paper, we have described a prototype of a Referent Tracking System which is able to maintain a large pool of data about particulars and their relations based on the Referent Tracking paradigm. The system is implemented to serve as a back-bone for

EHR applications either in a client server setting by means of web services or embedded in the EHR applications themselves.

The system maintains references to particulars and their relationships under the form of a RDF graph together with the information concerning which universals the particulars instantiate and the concept codes from the coding systems to which they are associated. By resorting to Basic Formal Ontology and the OBO Relation Ontology, and because of the referential semantics provided by the Referent Tracking paradigm, the data in the graph mirror the structure of reality. This set up paves the way to make machines understand EHR data unambiguously and is, we believe, an important contribution in reaching semantic interoperability.

The prototype is a first, though important, step towards deployment but much more work is required. Because EHR systems run under strict safety, security and confidentiality regulations, the RTS must follow the same principles. To protect the RTS against unauthorized access, we have a security module which is currently in an early stage of development. Thus far, the module only verifies a client on the basis of the user's user-name and password. Encryption and transaction certification (R. Housley, W. Polk, W. Ford, & Solo, 2002) will be dealt with later as well as improved access control concerning which parts of the graph are allowed to be accessed by a particular user.

REFERENCES

Bechhofer, S. (2003). *The DIG Description Logic Interface: DIG/1.1*. Paper presented at the Proceedings of DL2003 Workshop, Room, Italy.

Bechhofer, S., Horrocks, I., & Turi, D. (2005). The OWL Instance Store: System Description. In *Proceedings of the 20th International Conference on Automated Deduction* (Lecture Notes in Computer Science ed., pp. 177-181): Springer-Verlag.

Blobel, B. (2006). Advanced EHR architectures--promises or reality. *Methods Inf Med, 45*(1), 95-101.

Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., et al. (2004). Web Services Architecture, *W3C Working Group Note 11 February 2004*.

Borgida, A., & Brachman, R. J. (1993). Loading data into description reasoners. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (pp. 217-226): ACM Press.

Bouquet, P., Stoermer, H., Mancioppi, M., & Giacomuzzi, D. (2006). OKKAM: Towards a Solution to the "Identity Crisis" on the Semantic Web. In G. Tummarello, O.

Signore & M. Martinelli (Eds.), *Proceedings of SWAP 2006, the 3rd Italian Semantic Web Workshop* (pp. Electronic proceeding). Pisa Italy.

Broekstra, J., Kampman, A., & Harmelen, F. v. (2002). Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *Lecture Notes in Computer Science - International Semantic Web Conference ISWC2002* (Vol. 2342, pp. 54-68). Heidelberg: Springer.

Ceusters, W., Elkin, P., & Smith, B. (2006). Referent Tracking: The Problem of Negative Findings. In A. Hasman, R. Haux, J. v. d. Lei, E. D. Clercq & F. Roger-France (Eds.), *Studies in Health Technology and Informatics. Ubiquity: Technologies for Better Health in Aging Societies - Proceedings of MIE2006* (Vol. 124, pp. 741-746). Amsterdam: IOS Press.

Ceusters, W., & Smith, B. (2005). Tracking Referents in Electronic Health Records. In *Proceedings of MIE2005 - Medical Informatics Europe* (pp. 71-76). Amsterdam: IOS Press.

Ceusters, W., & Smith, B. (2006). Strategies for Referent Tracking in Electronic Health Records. *Journal of Biomedical Informatics, 39*(3), 362-378.

Ceusters, W., & Smith, B. (2007). Referent Tracking for Digital Rights Management. *Forthcoming in International Journal of Metadata, Semantics and Ontologies*.

Ceusters, W., Smith, B., & Flanagan, J. (2003). Ontology and Medical Terminology: Why Description Logics Are Not Enough. In *Proceedings of the Conference: Towards an Electronic Patient Record, May 10-14, 2003*. San Antonio: MA: Medical Records Institute (Electronic Publication).

Christensen, E., Curbera, F., Meredith, G., & Weerawarana, S. (2001). Web Services Description Language (WSDL) 1.1, *W3C Note 15 March 2001*.

FMA™ University of Washington. (2006). FMA™ (Foundational Model Anatomy Ontology). Retrieved 25th January, 2007, from http://sig.biostr.washington.edu/projects/fm/index.html

Foundation for Research and Technology – Hellas. (2003, July 18). The RDF Query Language (RQL). Retrieved 25 January 2007, from http://139.91.183.30:9090/RDF/RQL/

Gennari, J., Musen, M. A., Fergerson, R. W., Grosso, W. E., Crubezy, M., Eriksson, H., et al. (2003). The Evolution of Protégé: An Environment for Knowledge-Based Systems Development. *International Journal of Human-Computer Studies, 58*(1), 89 -123.

Grenon, P., Smith, B., & Goldberg, L. (2004). Biodynamic Ontology: Applying BFO in the Biomedical Domain. In D. M. Pisanelli (Ed.), *Ontologies in Medicine* (pp. 20-38). Amsterdam: IOS Press.

Haarslev, V., & Möller, R. (2001). Racer system description. In *International Joint Conference on Automated Reasoning, IJCAR2001* (pp. 18-23). Siena, Italy.

Health Level Seven Inc. (2007). Health Level 7. Retrieved 25th January, 2007, from http://www.hl7.org/

HP Labs Semantic Web Research. (2006). Jena- A Semantic Web Framework for Java. Retrieved 25th January, 2006, from http://jena.sourceforge.net/

JGraph Ltd. (2006). JGraph java API. Retrieved 25 January, 2007, from http://www.jgraph.com/

Kalyanpur, A., Parsia, B., Sirin, E., Grau, B. C., & Hendler, J. (2006). Swoop: A 'Web' Ontology Editing Browser. *Journal of Web Semantics 4*(2).

Kristol, D., & Montulli, L. (1997). *HTTP State Management Mechanism* (No. RFC-2109): Network Working Group.

Manola, F., & Miller, E. (2004). RDF Primer, *W3C Recommendation 10 February 2004*.

Manzoor, S., & Ceusters, W. (2007). Referent Tracking System. Retrieved 19 April 2007, from http://www.org.buffalo.edu/RTU/Shahid/RTS/

Mitra, N. (2003). SOAP Version 1.2 Part 0: Primer: W3C Recommendation 24 June 2003.

Mork, P., Brinkley, J. F., & Rosseb, C. (2003). OQAFMA Querying Agent for the Foundational Model of Anatomy: a Prototype for Providing Flexible and Efficient Access to Large Semantic Networks. *Journal of Biomedical Informatics, 36,* 501-517.

OBO-Edit Working Group. (2006). OBO-Edit: An Ontology Editor. Retrieved 25 January, 2007, from http://www.oboedit.org/index.html

Prud'hommeaux, E., & Seaborne, A. (2006). SPARQL Query Language for RDF. *W3C Working Draft 26 March 2007,* from http://www.w3.org/TR/rdf-sparql-query/

R. Housley, W. Polk, W. Ford, & Solo, D. (2002). *Internet X.509 Public Key Infrastructure* (No. RFC: 3280): Network Working Group.

RDF Data Access Working Group. (2007). ARQ - A SPARQL Processor for Jena. Retrieved 15th Febuary, 2007, from http://jena.sourceforge.net/ARQ/

Rosenbloom, S., Miller, R., Johnson, K., Elkin, P., & Brown, S. (2006). Interface Terminologies: facilitating direct entry of clinical data into electronic health record systems. *J Am Med Inform Assoc, 13*(3).

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., & Katz., Y. (2006). Pellet.

Smith, B., Ceusters, W., Klagges, B., Köhler, J., Kumar, A., Lomax, J., et al. (2005). Relations in biomedical ontologies. *Genome Biology, 6*(5), R46.

Smith, B., Kusnierczyk, W., Schober, D., & Ceusters, W. (2006). *Towards a Reference Terminology for Ontology Research and Development in the Biomedical Domain*. Paper presented at the KR-MED 2006, Biomedical Ontology in Action.

Smith, M. K., Welty, C., & McGuinness, D. L. (2004). OWL Web Ontology Language Guide, *W3C Recommendation 10 February 2004*.

SNOMED International. (2007). SNOMED-CT Codes. Retrieved 25th January, 2007, from http://www.snomed.org/

The Apache Software Foundation. (2005). Axis: A Webservices toolkit. Retrieved 25 January, 2007, from http://ws.apache.org/axis/

The Apache Software Foundation. (2006). Apache Tomcat Server. Retrieved 25 January, 2007, from http://tomcat.apache.org/

The Gene Ontology Consortium. (2007). The Gene Ontology. Retrieved 25 January, 2007, from http://www.geneontology.org/

The Internation DOI Foundation (IDF). The DOI System. Retrieved April 13, 2007, 2007, from http://www.doi.org/

Tsarkov, D., & Horrocks, I. (2006). FaCT++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)* (Lecture Notes in Artificial Intelligence ed., Vol. 4130, pp. 292-297). Heidelberg: Springer.

U.S. Department of Health & Human Services. (2006, January 11, 2007). International Classification of Diseases, Ninth Revision, Clinical Modification. Retrieved January 25th, 2007, from http://www.cdc.gov/nchs/about/otheract/icd9/abticd9.htm