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.

Categories and Subject Descriptors
H.3.5 [Information storage and retrieval] – Web-based services.

General Terms
Performance, Design, Reliability, Standardization, Languages, Theory

Keywords
Referent Tracking, Electronic Health Records, Ontology, RDF, Semantic Interoperability

1. 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 [16] or openEHR [3]. One particular component of a modern EHR deals with the access to terminologies and to coding and classification systems such as ICD-9-CM [39] or SNOMED-CT [35]. 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 [30].

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. But, certainly, existing EHRs do not contain enough information of the right sort to enable correct interpretation by software agents and thus to render different EHR systems semantically interoperable.

Here ontologies came into play, for which in the course of time various representation languages have been developed, the most recent one being the Ontology Web Language (OWL) [34]. In addition, there are tools such as Protégé [13], SWOOP [19] and OBO-Edit [25] which have been used for building ontologies such as the Foundational Model of Anatomy (FMA) [11] and the Gene Ontology [37]. Reasoning with such ontologies can be done with tools such as Pellet [31], Racer [15] and FaCT [38]. Thus far, however, none of these tools, neither the ontologies developed, are used in operational EHR environments, and this for several reasons. Many representation tools allow only class-level representations, while most current reasoners do not support reasoning over instances in ways that mirror the relationships between the instances in reality. These tools also typically fail when they are loaded with large amounts of instance data.

In this paper, we describe a software system which implements Referent Tracking, a paradigm designed to solve the problems just sketched. The system 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 system 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 system.

2. REFERENT TRACKING

2.1 Main Principles

Referent Tracking (RT) is a paradigm that was introduced in 2005 in the field of EHR systems and that is intended to avoid the ambiguities that arise when clinicians, when writing statements in an EHR, refer to entities on the side of the patients by means of generic terms [9]. RT does so by assigning globally unique IDs (called IUIs for: Instance Unique Identifiers) as 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.
Thus the information that is currently captured in the EHR analogous to sentences such as: “this patient has a left forearm fracture”, would need to be conveyed by means of descriptions such as “#IUI-5 is located in #IUI-4”, together with associated information to the effect that “#IUI-4” refers to the patient under scrutiny, and “#IUI-5” to a particular fracture in patient #IUI-4 (and not to some similar left forearm fracture from which he suffered earlier).

Information of this sort is stored in a Referent Tracking System (RTS). The purpose of an RTS is, as its name suggests, to keep track of referents which are entities that exist 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 forearm, the specific fracture in John’s forearm, and so forth. These particulars are instances of universals such as person, 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 that adhere to the principles laid down in Basic Formal Ontology (BFO) [14] on which RT further builds, by means of classes, but with the additional constraint, in contrast to other approaches to ontology building, that classes only refer to universals. Therefore, an RTS must correspondingly also contain information relating particulars to classes, such as “#IUI-5 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 [33], a configuration of particulars and or universals is called a portion of reality (POR).

In [7, 8] the conditions for assigning an IUI to a particular are described, as well as the templates according to which some PORs are to be represented in an RTS. 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.

### 2.2 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. Therefore, an RTS should be able to:

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

| Table 1 Abstract syntax and semantics of information templates in a referent tracking system |
|------------------------------------------|----------|----------|
| **Template Name** | **Abstract Syntax** | **RDFS class** |
| A-template | $\text{IUI}_a = \langle \text{IUI}_a, \text{IUI}_t, \text{t}_a \rangle$ | Particulars
| PtoU-template | $\text{U}_p = \langle \text{IUI}_p, \text{t}_p, \text{r}, \text{o}, \text{IUI}_a, \text{u}, \text{t}_a \rangle$ | PtoU
| PtoCo-template | $\text{C}_o = \langle \text{IUI}_o, \text{t}_o, \text{cbs}, \text{IUI}_a, \text{co}, \text{t}_a \rangle$ | PtoCo
| PtoN-template | $\text{N}_n = \langle \text{IUI}_n, \text{nt}, n_i, \text{IUI}_o, \text{t}_o \rangle$ | PtoN
| PtoLackU-template | $\text{U}_p = \langle \text{IUI}_p, \text{t}_p, \text{r}, \text{o}, \text{IUI}_a, \text{u}, \text{t}_a \rangle$ | PtoLackU
| Meta-template | $\text{D}_d = \langle \text{IUI}_d, \text{X}, \text{t}_d \rangle$ | Meta

Publication of a description of a portion of reality in the RTS where \text{IUI}_a is the IUI of the entity registering \text{X} in the system, \text{X} is the information-unit in question (in the form of any other template above), and \text{t}_d is a reference to the time the registration was carried out.
3. APPLIED TECHNOLOGIES

3.1 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 classes and the 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 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 class.

3.2 Resource Description Framework
In a statement such as “John (#IUI-1) has a fracture (#IUI-6) 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-6 depends_on #IUI-3, form the edges in the graph. Therefore, the Resource Description Framework (RDF) [21] 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. An RDF statement can be represented as a directed graph, where the subject and objects are nodes and the predicate is a directed arc pointing from the subject to the object. For example, an assertion that the particular rts:IUI-1 was seen (rts:iuia) by the physician rts:IUI-1 and that the creation time (rts:tap) is 01/12/2006 can be represented by two RDF statements as shown in Figure 1. In RDF, eclipse shapes are used to represent resources and the rectangular shapes represent atomic values.

Our RDF representations of the RT templates are treated as resources themselves: each 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 such that for instance the resource rts:IUI-1 is the same as http://org.buffalo.edu/RTS#/IUI-1.

Figure 1: RDF graph representation

An assertion to the effect that particular #IUI-1 has name John can be represented in RDF as shown in Figure 2 by means of three resources: two first order resources for the particulars rts:IUI-1 and rts:IUI-10, and one second order resource for the PtoN template (rts:pton_2). The RDF triple (rts:pton_2 rts:iuia rts:IUI-1) in the RTS denotes that the PtoN template resource rts:pton_2 associates the name John to the particular rts:IUI-1.

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 classes and their properties and relations. We have mapped the RT templates to RDFS classes, thereby ensuring that the class names are identical to the template names, with the exception of PtoU-, which, because of restrictions in the RDFS naming conventions, has been mapped to PtoLackU.

Figure 2: RDF graph for rts:IUI-1 has name John

We have defined the RT URI ‘rts:type/terminologysystemid/termid’ for the external resources denoting universals, concepts and relations. The RT URI starts with the rts: prefix and the type part represents whether this URI represents a universal, a concept, or a relation. The possible values for type are u for universal, cbs for concept code and r for 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.

Figure 3, an extension of Figure 2, shows that particular rts:IUI-1 has the name John and that John enjoys the has_part relation (taken from the OBO_REL ontology [32] with particular rts:IUI-3 (the particular rts:IUI-3 is an instantiation of the FMA class Left forearm). The figure contains resources with property rdf:type indicating of what class these resources are instances, as well as resources for which no RDFS class has been defined. In the figure, the rts:iuia property of rts:pton_5 resource (a particular PtoP instantiation) 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 :IUI-1 (John) has_part rts:IUI-3 (instance of the Left forearm).

4. RTS ARCHITECTURE

We have designed the RTS as a server application as well as a Java library. All materials to do so are downloadable from Sourceforge under an open source license [22].

The architecture of the application is shown in Figure 4. 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 4: RTS Architecture](image)

4.1 RTS Web Services

Web services are remote procedures hosted at an http server which are invoked through SOAP messages [23] which contain the procedure information (procedure name, parameters and return type) and port type (location of the procedure). The RTS uses Axis for Java [36] to host the web services thereby taking advantage of the native support of the Web Services Definition Language (WSDL) [10] that Axis provides. Both WSDL and SOAP are platform independent so that the RT interface becomes accessible to all programming platforms and environments.

The RTS web services allow both retrieval and insertion of the RT templates in the RTS. They run over the http protocol [4] 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 [20]. 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.

4.2 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.

4.2.1 RTRepository

The RTS has been built to be independent of any data source technology. To achieve this goal, we have defined the RTRepository class as an abstract Java class. This class provides all necessary services for managing the data based on the principles defined in the RT paradigm. 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 class by extending the RTRepository such that it targets the SAIL Sesame API for manipulating RDF graphs as a data source [6].

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 and fast access purposes.

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 (a file maintains the different configurations of the RTS such as initialization parameters). This allows more 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. An abstract view of the mapping of the RT data into java classes is shown in an UML diagram (Figure 5). The RTRepository consists of the instances of the Resource interface where each resource has a unique id. There are two types of resources at this level: ParticularRepresentation and Relation. The name convention used in the java mapping is the same as discussed in the data representation section.

![Figure 5: RTRepository API abstract view](image)
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 new RT template resources in the repository. The most basic service assigns a IUI to a real world entity and creates its representation in the RTS. The next step is to assign detail to this particular. For example, the code

```
'PtoU ptou = repository.createPtoU(particular.getIUI(),
iuia,"instance_of", "FMA", "Left forearm", ta, tr);
```

relates a particular created earlier to the *Left forearm* class of the FMA by means of the *instance_of* relation.

Importantly, the RT paradigm does not allow any delete 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 template resources in the **RTRepository**, the resources are cached right after the create operation. The client can remove or modify resources 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).

All arguments in the above services can be null, but not at the same time. Because the search pattern in the services might match with several thousands of particulars and the network bandwidth might not allow to transfer that many results to the clients, we have set the limit by default to return the first 200 resources. 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). 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.

### 4.2.2 Querying the RTS using SPARQL

Because the RT data are expressed in RDF, RDF query languages such as RQL [12], SPARQL [27] and SeRQL [6], 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) [29]. 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.

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 ?ptou 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 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:pptop_1", i.e. the *URI* of the matching resource.

**Listing 1** enumerates the triples involved in representing that #IUI-3 (*John’s left forearm*) is part of #IUI-1 (*John*).

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

<table>
<thead>
<tr>
<th>Service Name</th>
<th>Service Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getParticulars</td>
<td>This service retrieves the particulars and the associated <em>PtoN</em> templates.</td>
</tr>
<tr>
<td>WithPtoN (iuip, nt, n, iuia, taRange, tdRange)</td>
<td>The query <code>getParticularsWithPtoN (null, &quot;name&quot;, &quot;John&quot;, null, null)</code> (which particulars have the name John) will for the data shown in Figure 2 retrieve the resources rts:ptop_2 and rts:IU1-3.</td>
</tr>
<tr>
<td>getParticulars</td>
<td>This service retrieves the particulars which do not have an associated <em>PtoN</em> templates.</td>
</tr>
<tr>
<td>WithPtoCo (iuip, co, iuia, taRange, tdRange)</td>
<td>The query <code>getParticularsWithPtoCo (null, &quot;rts:co//SNOMED-CT/91419009&quot;, null, null, null)</code> retrieves the particulars annotated with the SNOMED-CT code '91419009', which is a code for <em>Left forearm fracture</em>.</td>
</tr>
<tr>
<td>getParticulars</td>
<td>This service retrieves the particulars which are instances of the universal <em>u</em>.</td>
</tr>
<tr>
<td>WithPtoU (iuip, u, iuia, taRange, tdRange)</td>
<td>The query <code>getParticularsWithPtoU (null, &quot;rts:w/FMA/Forearm&quot;, null, null, null)</code> retrieves the instances of the FMA class denoting the universal <em>Forearm</em>.</td>
</tr>
<tr>
<td>getParticulars</td>
<td>This service retrieves the particulars which do not stand in any <em>lacks</em> relation to the universal <em>u</em>.</td>
</tr>
<tr>
<td>WithPtoLackU (iuip, u, iuia, taRange, tdRange)</td>
<td></td>
</tr>
</tbody>
</table>

### Listing 1: Triple View of the RDF

```
1. rts:IUI-1 rdf:type rts:Particular
2. rts:IUI-1 rdf:tap "28/08/2006"
3. rts:IUI-3 rdf:type rts:Particular
4. rts:IUI-3 rdf:tap "28/08/2006"
5. rts:ptop_5 rdf:type rts:PtoP
6. rts:ptop_5 rts:iuip rts:IUI-3
7. rts:ptop_5 rts:u rts:IUI-4
8. rts:ptop_5 rts:r rts://OBO_REL/has_part
9. rts:ptou_5 rdf:type rts:PtoU
10. rts:ptou_5 rts:iuip rts:IUI-4
11. rts:ptou_5 rts:u rts://FMA/Left+forearm
12. rts:ptou_5 rts:ta "28/08/2006"
13. rts:ptou_5 rts:tr "28/08/2006"
```

The query
```
1. SELECT ?ptou
2. WHERE {
4. }
```

requests the resources (*PtoU* templates) which are related to the universal *Left forearm*. 
In SPARQL more complex queries can be built by adding more triple pattern restrictions. For example, the query

```sql
1 SELECT ?ptou ?p ?ptop
2 WHERE{
4   ?ptop rts:r //OBO_REL/has_part
7   ?ptou rts:u  rts:t/./FMA/Left+forearm.
8 }
```

requests the particulars that are instances of Left forearm and a part of particular rts:IUI-1. 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 (described further down) which upon execution returns the results. The results are the URI for the RT templates resources. The repository, in the second step, then queries the Data Access API (described further down) to retrieve all the attributes of the returned resources 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.

### 4.2.3 RTVisGraph

This component is an extension of the JGraph java library for displaying graphs [18] which has the ability to generate images in Jpeg and SVG. 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-6 (annotated by SNOMED fracture code).

### 4.3 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 [17] for RDF manipulation which does the same job as SAIL might be another choice. SAIL comes with a RDFRepository java interface, which represents a logical data repository for RDF graphs.

As a further improvement, rather than using the implementations of the SAIL API directly, we have written the RDFRepositoryWrapper (Figure 6) over the implementations of SAIL. The purpose of the wrapper is to call under certain circumstances the reasoning services described below.

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.

### 4.4 Reasoning API

Reasoning is a core part of the RTS and its purpose is double: first to avoid inconsistent data to be 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 [24], some coming with a DIG interface [1] 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 so called OntologyConnector which provides an interface to the external ontology systems. The OntologyConnector interface services (as shown in Table 3) are designed based on the principles defined in the OBO Relation Ontology [32] and Basic Formal Ontology [14]. 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 ontology, 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.

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 is caching 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/Left+forearm)
```

is executed over the data of Figure 3, 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 resource. Finally it requests the ontologies in which the universals are represented by calling the isSubsumedBy(“Left forearm”, “Forearm”) service from the OntologyConnector instance of the specialized class implemented for the FMA ontology. If subsumption can be applied, then it returns the resulting particulars with their associated templates.
5. 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 [32]. 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 the services as getParticularsWithPtoPByPtoU, getParticularsWithPtoN, getParticularsWithPtoPByPtoCo and getParticularsWithPtoPByPtoU. 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. Table 4 compares the retrieval times in milliseconds for RDFSRepositoryRTNativeImp (RT native RDBMS RDF repository) and RDFSRepository (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.

6. CONCLUSION & FUTURE WORK
In this paper, we have described a first 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 security, privacy 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 [28] 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. Also further comparison of its components with other available tools that have similar objectives is required, e.g. Instance store which is capable of maintaining a large pool of instances [2]. It is however restricted to OWL-DL based reasoning over classes and does not provide any mechanism to search for instances based on their relations. Other applications providing reasoning over instances are [5, 26].

<table>
<thead>
<tr>
<th>Service Name</th>
<th>Service Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isUniversalExist(u):</td>
<td>This service checks whether a universal u exists in the ontology system.</td>
</tr>
<tr>
<td>isUniversalSubType(u1, u2):</td>
<td>This service checks whether the universal u1 is a subtype of universal u2.</td>
</tr>
<tr>
<td>isRelationExistBetweenUniversals(r, u1, u2):</td>
<td>This service checks whether the relation (r) exists between universals u1 and u2.</td>
</tr>
<tr>
<td>isUniversalSubsume(u1, u2):</td>
<td>This service checks whether the universal u1 is subsumed by universal u2</td>
</tr>
<tr>
<td>getRelations(u1, u2):</td>
<td>This service returns the list of the relations that exist between two universals u1 and u2.</td>
</tr>
</tbody>
</table>

Table 4: OntologyConnector class services

<table>
<thead>
<tr>
<th># of RT Templates</th>
<th># of Particulars</th>
<th>Query set execution time in milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>162552</td>
<td>51706</td>
<td>RT native persistence</td>
</tr>
<tr>
<td>350075</td>
<td>111300</td>
<td>Sesame native persistence</td>
</tr>
<tr>
<td>540430</td>
<td>171818</td>
<td>RT Native persistence</td>
</tr>
<tr>
<td>788143</td>
<td>250663</td>
<td>Sesame RDF repository</td>
</tr>
<tr>
<td>1279908</td>
<td>406360</td>
<td>RT Repository</td>
</tr>
</tbody>
</table>

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.

7. REFERENCES


[18] JGraph Ltd. JGraph java API, 2006.


