LORI: Linguistically Oriented RDF Interface For Querying Fuzzy Temporal Data

By

Majid Robatjazi

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

In

Software Engineering and Intelligent Systems

Department of Electrical and Computer Engineering
Edmonton, Alberta

Abstract

The concept of Semantic Web, introduced by Berners-Lee in 2001, emphasizes importance of expressing semantics of data stored on the web. The introduced data format called Resource Description Framework (RDF) is a meaningful way of expressing and exploring data relations. It provides basics for constructing semantics-oriented data formats including ontology. More and more often ontology and RDF are used to represent variety of data including N-ary relations and temporal information. On many occasions this results in complex data structures. Their utilization requires a full understating of used data configurations.

The thesis introduces and describes a methodology for querying RDF-based data containing temporal information and built using non-trivial data structures. The methodology is suitable for dealing with logic-based data structures – ontology – and less constrained data formats – RDF. A significant contribution of the thesis is a fuzzy-based Linguistically Oriented RDF Interface – LORI. The interface includes specialized build-in predicates suitable for constructing temporal queries and supporting imprecise phrases describing time and data features, and high-level predicates built based on them. A number of case studies focused on querying time-based events, as well as and their performance evolutions are presented.

This thesis is dedicated to my wife, father and the memory of my mother.

Acknowledgement

I would like to express my sincere gratitude to my supervisors Dr. Marek Reformat and Dr. Witold Pedrycz for the continuous support of my M.Sc. study and related research, for their patience, motivation, and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis. I would also like to thank Dr. Peter Musilek and Dr. Stanislav Karapetrovic for agreeing to serve on my examination committee. And finally, my special gratitude goes to the Department of Electrical and Computer Engineering of University of Alberta for its endless friendly support and assistance.

Table of Contents

List of Tables

List of Figures

List of Symbols

API - Application Program Interface

DBLP - Digital Bibliography & Library Project

FBTL - Fuzzy Branching Temporal Logic

FLTL - Fuzzy Linear Temporal Logic

FTL - Fuzzy-time Temporal Logic

GUI - Graphical User Interface

HTTP – Hyper Text Transfer Protocol

HTML – Hyper Text Markup Language

JSON - JavaScript Object Notation

LOD - Linking Open Data

LORI - Linguistically Oriented RDF Interface for Querying Fuzzy Temporal Data

OWL - Ontology Web Language

OWL DL - OWL Description Logics

RDF - Resource Description Framework

RDFS - Resource Description Framework Schema

SPARQL - SPARQL Protocol and RDF Query Language

SWRL - Semantic Web Rule Language

URL - Uniform Resource Locator

URI - Uniform Resource Identifier

XML - Extensible Markup Language

1. Introduction

1.1. World Wide Web – State of the Art

The important concepts of Semantic Web [44] and Lined Open Data [16] – perceived as the prelude to Web 3.0 – are associated with a data representation format called Resource Description Framework (RDF) [22]. The significance of RDF comes from its ability to represent semantics of data in a form of relations existing between pieces of information. It can be said that RDF data constitutes an important step towards creating a foundation for methods and approaches leading to a more intelligent and human-oriented way of processing, analyzing and utilization of any data and information.

There is an increasing trend of representing 'richer' information that contains multifaceted features and relations, as well as temporal information attached to them. The consequence of that is an introduction of not-trivial data structures. At the same time the users' expectations regarding easiness and effectiveness of 'interaction' with data is growing. The users would like to see more human friendly ways of asking for relevant information. It seems very reasonable to say that with an increasing amount of data it is difficult for the users to ask questions with precisely identified quantitative and temporal values of data. Also, RDF that is not inherently suitable for expressing N-ary relations, and the proposed solutions [13] introduce complexity and difficulties in data processing and analysis.

There is a growing demand for systems that are able to incorporate semantics in processing and analyzing data. Increasing popularity of ontology as defined in the context of the Semantic Web makes such tasks reasonably feasible. Ontology provides the ability

to use hierarchy of concepts, their definitions and relations, as well as rules defined based on these concepts in any domain of interests.

Many real-world applications involve processing of temporal data. The temporal data and terms used to describe temporal patterns and information are quite often expressed in an approximate manner. This is especially visible in the case of knowledge provided by human experts who use imprecise terms describing qualitatively and temporally variety of facts, and building rules based on these facts.

1.2.Objectives and Contributions

In the context of the new developments related to advanced and semantic-oriented data representation formats on one side, and the users' expectations of accessing and processing complex information and data on the other side, there is demand for mechanisms and tools addressing the users' needs.

This thesis is an attempt to equip new data representation forms with mechanisms providing the user with the ability to represent temporal and approximated – fuzzy – information. Here, we propose simple fuzzy and temporal ontologies containing basic concepts and relations that can be treated as a framework for building knowledge bases capable of representing and processing fuzzy temporal data. In order to enable reasoning with such data a number of built-in predicates have been designed and implemented.

We introduce a Linguistically Oriented RDF Interface – LORI – that provides the users with the ability to exploit temporal data containing complex relations. LORI eliminates needs for an extensive knowledge of details related to the structure of queried data, and allows for using imprecise expressions built with quantitative and time-based terms.

The processes of designing and developing ontologies, ontology-based mechanisms for querying data, and the query interface – LORI – embrace multiple aspects of dealing with temporal and complex data, and results in the following realizations:

- designing and developing ontologies capable of representing temporal as well as fuzzy information;
- developing temporal predicates as built-in functions of Jena's that can be utilized as atoms during a process of constructing ontology-based IF-THEN rules in SWRL (Semantic Web Rule Language);
- designing an architecture of LORI based on an idea of two interfaces: 1) low-level one called *ReasonerInterface* which provides necessary rules and predicates to deal with temporal and complex data, built on Jena's RDF/RDFS reasoner; and 2) *UserInterface* composed of high-level predicates and built by data expert based on *ReasonerInterface*;
- developing temporal predicates as built-in functions of Jena's RDF/RDFS reasoner; these predicates utilize fuzzy terms to express imprecise declarations of time;
- proposing and developing a data structure for storing query results, with the ability to create sequences of queries, as well as to store, process and merge individual results;
- identifying a flexible approach for mapping high-level queries to low-level ones; the proposed idea is based on a mapping file that allows for dynamic changes and modifications of mapping rules.

1.3. Thesis Outline

The thesis is organized in the following way. Chapter 2 contains generic background and related work. Such basic concepts as Linked Data, Semantic Web, and ontology … are introduced and briefly described. A subsection is dedicated to work done by others on the topics of fuzzy temporal data representation and reasoning. Chapter 3 is dedicated to the first phase of the work – representing temporal and fuzzy data in ontology and querying it. The details of ontologies capable of representing fuzzy data as well as temporal data are provided. All developed predicates that allow for constructing rules for querying data represented in ontologies are described. A simple illustrative

example how the proposed techniques can be used is included. The results of continuation of our work are shown in Chapter 4. This chapter includes detailed description of the proposed and developed method for querying fuzzy temporal data containing complex data structures. The proposed method uses Resource Description Framework (RDF) as data representation. A description of LORI – Linguistically Oriented RDF Interface is included tougher with simple example of its utilization. Chapter 5 contains implementation details and more experimental results. Part of the chapter is dedicated to details of software libraries and packages that have been used in implementation of LORI, while the other part includes descriptions of the results of testing and validation processes preformed on the implemented LORI. The chapter related to contributions and future work concludes the thesis.

2. Generic Background and Related Work

2.1. Linked Data

The World Wide Web has radically changed a way people disseminate and share data and information. People simply publish texts and documents on global information space, and create links between them for the purpose of exchanging knowledge. The amount of data stored on the web has grown rapidly. Specialized applications and systems are required to explorer, utilize and exchange information. The application of tagged information, meta data, markup languages and XML, has turned the web information into a more understandable and expressive data suitable for reading not only by humans but also by machines.

Recently, the Internet has changed from a linked-documents space to a linked-documents and data space. The connected structured data on the Web is known as Linked Data [4]. Understanding of current Web, Web of document, is essential as general architecture of World Wide Web is applied to make and sharing structured data which is known as Web of Data. The HTTP protocol, URIs, HTML and Hyperlinks are main used principles in the Linked data and Web of Data. Linked Data lay on the same architecture principles of the Web of documents. [11]

This new Web is suitable for system agents and machines to exchange information and knowledge. Linked data relies on RDF, Resource Description Framework, as a standard to publish and link things in the different Web resources includes documents. It is simple and appropriate for current Web architecture.

URL, Uniform Resource Locator, is address of documents in the web of documents and URI, Uniform Resource Identifiers, is address of entities and things in the Web of data.

Both URLs and URIs rely on HTTP, Hyper Text Transfer Protocol, which is fundamental universal mechanism for retrieving resources. URI, HTTP and RDF, Resource Description Framework, are main technologies that Linked Data relies on them. So the Web of Data can be considered as a new layer on the classic Web of documents.

The URI identifies things on the Web of data and it is suitable for identifying things not only because it is a simple method to make a universal unique name but also it is accessing information describing to things.

Principles that have been introduced by Tim Berners-Lee for publishing data on global data space are following [11]:

- User URIs as name for things
- User HTTP URIs so that people can look up those names
- When someone looks up a URI, provide useful information
- Include links to other URIs, so that they can discover more things

Publishing data in RDF format has been increased during the past years and lots of groups, organizations and individuals have agreed to use Linked Data as a new way to publish data. There is an open project under W3C, which is called Linked Open Data Community. Its goal is to extend the Web with data commons by publishing various open RDF datasets. These datasets are published on the web and by setting RDF links between data items from different data sources. The number of triples on the project was over two billion RDF triples on October 2007. This number had grown to 31 billion RDF triples by September 2011, [17]. Table 1 shows the amount of datasets that have been published on the Web of Data. Fig. 1 depicts the relations between 12 RDF datasets as of 2007, while Fig. 2 depicts the relations between 570 RDF datasets in 2014.

Table 1 - Number of datasets in LOD

Date	# Datasets
May 2007	12
Sep 2008	34
Jul 2009	95
Sep 2010	203
Sep 2011	295
Aug 2014	570

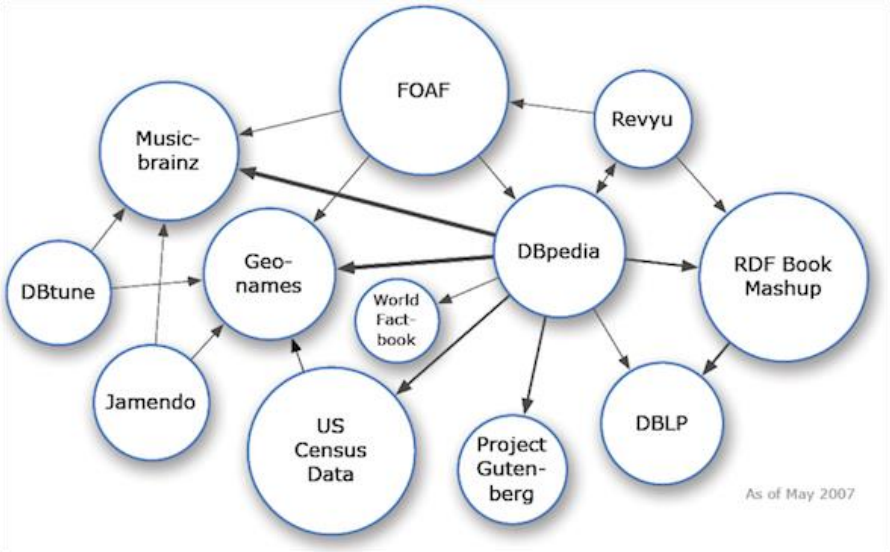


Figure 1 - Relation of 12 RDF datasets on 2007

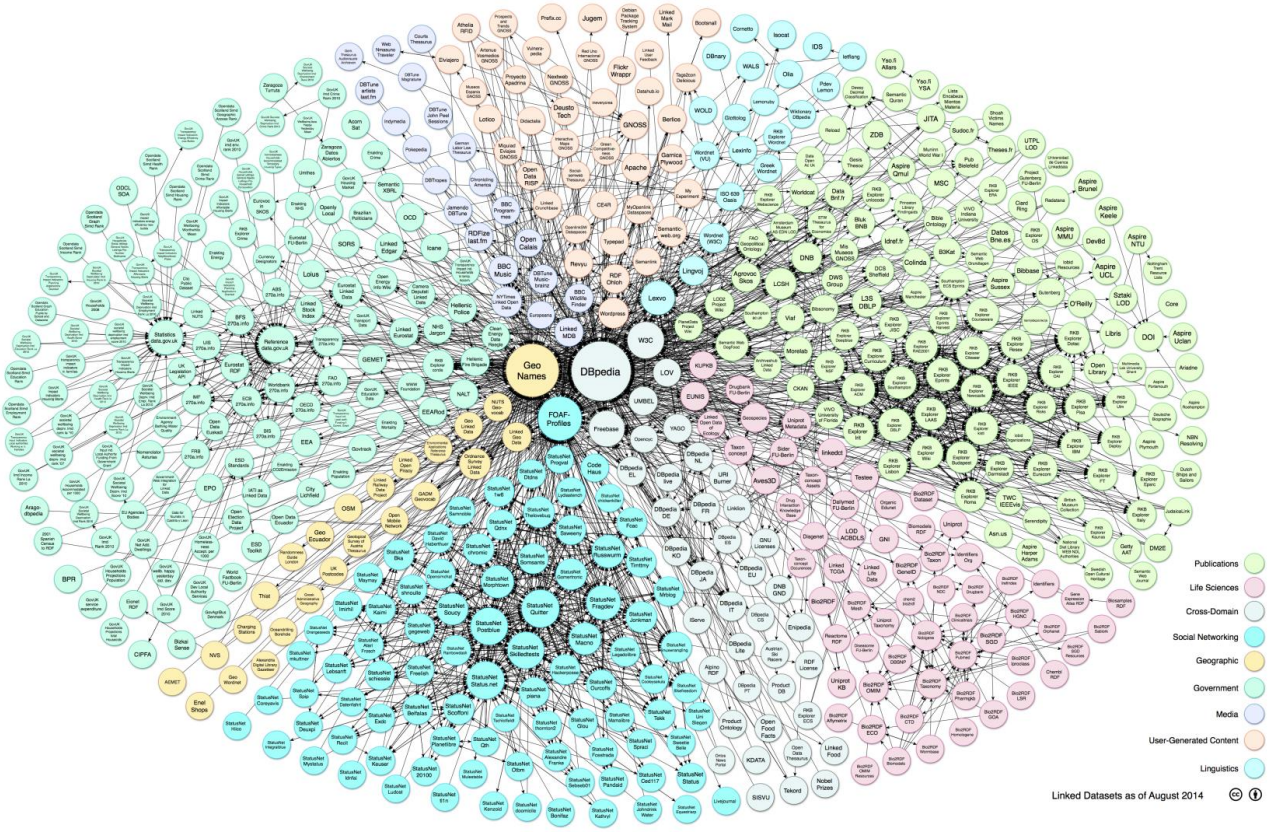


Figure 2 - Relation of 570 RDF datasets in 2014

2.2. Semantic Web

The Semantic Web [44] also referred to Web of Data, which introduced by Tim Berners-Lee and developed by W3C is stacking of technologies that helping to build web of data. Although there are different interpretations for goal of Semantic Web, majority of original literature agreed constructing of machine readable global Web is the main goal for Semantic Web. The web of documents is intended to be used by human and machines use it under supervision of human. In semantic web, machines can understand and act upon data [11]. Constructing a global Web that not only human can understand it but also machines can understand it naturally is the first step of creating Semantic Web [43].

To reach this goal converting web of unstructured documents to web of structured data is essential. In this process Linked Data playas the main role to reach this goal [11]. It leads to make a new space that users and machines can publish, find and share data more easily.

An example of World Wide Web usage is search engines that are important for users to fine information among global space. Despite lots of improvement on search engines during these days, they have serious problem for finding proper result as they are keyword base search tools, [9]. Low or no recall that user gets no result or not getting relevant documents. High recall and low precision that user gets relevant documents among lots of irrelevant documents which makes difficult for user to find proper information. Current search engine result is highly sensitive to keywords that users are using for search and if relevant documents use different terminologies then it leads missing relevant documents. Result of search engines are single documents and if relevant information is spread over different documents then user should find all relevant documents and extract information and put them together.

Semantic Web development is based on a stack approach, Fig 3, which each step is built on top of another step. Each layer is aware of lower layer and can interpret its information and also each layer is partially aware of higher layer, [9]. Resource Description Framework (RDF), Web Ontology Language (OWL) and Extensible Markup Language (XML) are language that is specifically designed for data.

Semantic web technologies are stacked and combined to replace content of web of documents. These are used for create link between data in which machines can process knowledge among relations instead of text processing. Machines capture the meaning of information not by specifying its meaning but by specifying how information interacts with other information. In this way machine can mimic human deductive reasoning and inference.

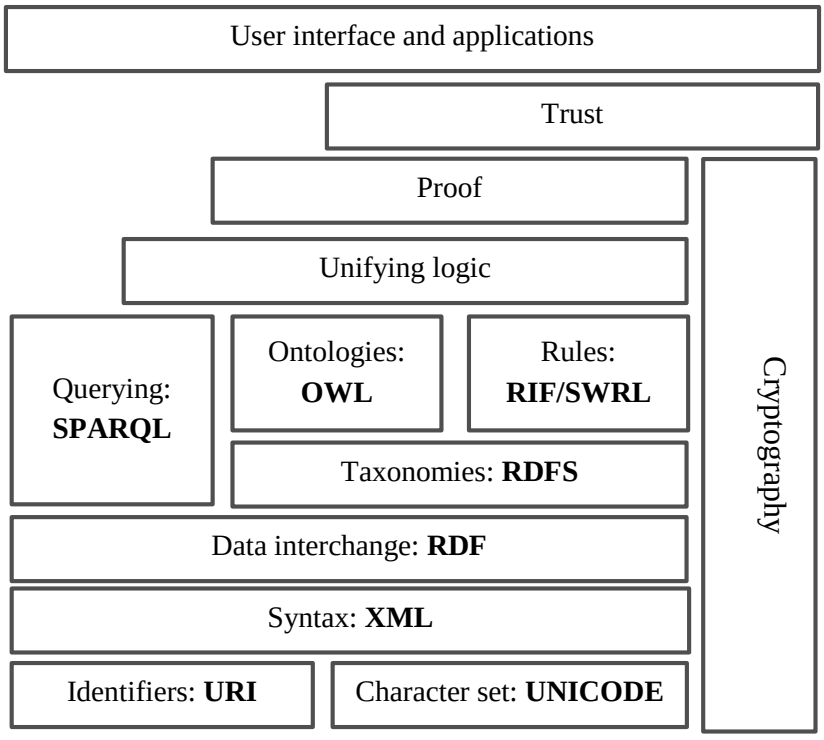


Figure 3 - Semantic Web stack

2.3. Ontology and Ontology-based Rules

The term “ontology” originally came from Greek work and it is study of nature of existence of things and their descriptions in philosophy. The term “ontology” is borrowed from philosophy by computer science. There is difference of meaning for ontology in computer science and the term “an ontology” is used instead of “ontology”. The most popular definition of an ontology, in the context of the Semantic Web [44], is “an explicit and formal specification of a conceptualization of a domain of interest” [45].

For publishing data in a common global space, having a common terminology is essential. Different datasets in Web of Data need to understand each other and this semantic interoperability is possible with ontology. This corporation is either with mapping terminologies to a shared ontology or direct mapping of ontologies. Currently following ontology languages are most significant; RDF as data model for objects and relations between them, RDF Schema as a vocabulary description language and OWL that is richer vocabulary description language by compare to RDF and RDF schema. OWL is an extended vocabulary language for defining disjointness on concepts (classes), cardinality, equality, richer typing of properties, enumerated classes and characteristics of properties.

Although distribution of understanding of information for people and machines is the most common goal of building ontology, there are some other goals for that, [45]. Reusing of domain knowledge recently became another important goal for ontology; one can use existing ontology for a particular common used domain, for example time, and extend it. Another goal for making ontology is to make domain assumptions explicit; it makes changes on our assumption of domain of interest easy if our knowledge about it changes.

Ontologies are more than just a vocabulary; they are sources of knowledge of a specific domain. Currently, most of ontologies are implemented in OWL (Web ontology language) that is based on RDF and designed by W3C.

The most important aspect of ontologies used for Semantic Web applications is related to identifying two ontology layers: the ontology definition layer, and the ontology individual layer. The ontology definition layer represents a framework used for establishing an ontology structure – based on *is_a* relation between classes – and for defining classes (concepts) existing in a given domain. The ontology individual layer, on the other hand, contains concrete information as instances of defined classes.

The ontology definition contains descriptions of all classes of an ontology. The classes are defined using datatype properties and object properties. These properties are:

- the *datatype property* focuses on describing features of a class; datatype properties can be expressed as values of data types such as Boolean, float, integer, string, and many more (for example, byte, date, decimal, time);

- the *object property* defines other than is-a relations among classes (nodes); these relations follow the notion of the RDF that is based on the triple subject- predicate-object, where: subject identifies what object the triple is describing; predicate defines the piece of data in the object a value is given to; and object is the actual value of the property; for example, in the triple John likes books, John is the subject, likes is the predicate and books is the object.

Both types of properties are important for defining ontologies. The possibility of defining class properties and relations between classes creates a versatile framework capable of expressing complex situations with sophisticated classes and the multiple different kinds of relationships existing among them. Once an ontology definition is constructed, its instances, called individuals, can be built. The properties of classes are filled out: real data values are assigned to datatype properties, and links to instances of other classes (individuals) are assigned to object properties.

It has been identified [27] that the OWL has limitations in the case of representing relations between complex properties. This has been overcome by putting together OWL and a rule language. As the result of that, the Semantic Web Rule Language (SWRL) has been introduced [27] [35] as a combination of OWL with RuleML (the sub-language of Rule Markup Language).

In SWRL, a rule axiom consists of an antecedent (body) and a consequent (head). The basic element of both antecedent and consequent is an atom. SWRL defines five basic atoms that can be used to build a rule:

- *C(x)* it is the simplest atom, it is used to check if a given instance x is the instance of concept C, for example, Person(John) represents an atom that checks if John is the instance of the concept Person;

- *P(x,y)* it is the atom that allows for checking if two instances x and y are related to each other via a property P, for example, liveIn(John, Edmonton) is ”looking” at the existence of the property liveIn between the instances John and Edmonton;

- *Q(x,z)* it is the atom that verifies if a data property Q of instance x has a value z, for example, *lastName*(John, Smith);

- *sameAs(x,y)* holds if instances (individuals) x and y are the same;

- *differentFrom(x,y)* holds if instances (individuals) x and y are different.

All atoms presented above can be used with variables instead of instances (x, y) and values (z). In this case, the atom P(x,y) can be used in the following way - liveIn(?a, Edmonton), and it would represent a question: who lives in Edmonton? Using the SWRL together with an ontology, it is possible to build rules based on object properties of the concepts defined in this ontology.

In summary, one of the most significant technologies for supporting the sharing, integration and management of information sources in knowledge base systems is ontology. In particular, the Ontology Web Language and its associated Semantic Web Rule Language [12] provide a powerful standardized approach for representing information and reasoning with it.

There are some steps for creating an ontology for a domain of interest. First of all defining ontology domain and scope; in the sense of what is the purpose of ontology and what type of question it should provide answers. The next step is defining existing concepts and concept hierarchy in the domain; there are 2 main approaches for creating hierarchy: top-down and bottom-up. In the first approach most general concepts are defined first and in the second approach the most specific concepts are defined first. The next step for creating an ontology is defining properties of each concept and their specifications and restrictions, for example value type, allowed values or cardinality. During the process of creating an ontology it is important to consider existing ontology related to domain of interest.

It is important to know that there is not just one correct way to create an ontology model. The best model is depending on the purposes of model and future extension of the model. Also creating an ontology model is an iterative process and it evolves during ontology development. Another tip for creating an ontology model is that concepts are usually close to physical and logical objects; if someone describe domain of interest in sentences then concepts are most likely to be nouns and verbs.

Creating an ontology itself is not a goal. It is developed to define a data structure and terminology for other software and agents to use. An example of ontology model is Wine ontology. It is definition of wines and foods and suitable combination of them. The wine ontology can be used by another software to make wine suggestions for menu of the day [34].

In the wine ontology, two main concepts are Wine and Food. They have sub-concepts as Fig. 4 depicts:

Figure 4 - Ontology hierarchy of Wine and Food

2.4. RDF / RDF Schema / SPARQL

A standard content format is an essential agreement on Web of Data. This standard is important for publishing data as all different applications, agents and machines need to process this content and communicate based on it. In the current web XML, Extensible Markup Language, is common widely used to represent and describe data. Although it is easy and simple to use for exchanging data and also it is general framework for exchanging of data between applications over the Web, it is not suitable for Web of Data as it does not provide meaning for data [9].

The concept of Semantic Web [44] has introduced a principal format of representing data called Resource Description Framework – RDF [22]. A building block of RDF is a simple triple: subject-predicate-object, where: subject identifies an entity the triple is describing; predicate defines a type of relation that exists between the subject and object; and object is an entity or a value describing the subject via being in relation with it. For example, in the triple

```
John travel Tokyo
```

John is an entity that is being described; travel is a relation that exists between the entities; and Tokyo is a value of this relationship. In other words we say that John travel(s|ed) to Tokyo. Further, a subject of one triple could be a subject of other triples. For example, the two following triples:

```
John type Person;     John birth 1996
```

Indicate that John is a person, and he was born in 1996. Additionally, an object of one triple could become a subject of another triple, or a subject of one triple can be an object of another. Overall, multiple entities can be involved in different relations and play different roles in these relations. That leads to a highly interconnected network of related entities.

RDF triples can be used to represent any type of information in any domain. There are a number of initiatives focusing on building repositories of predicates called

vocabularies [18] [20]. Descriptions of all RDF components are contained in RDF Schema [24].

RDF link triples, that object of triple is a URI, is used for making link between RDF triples. Power of RDF is that one can make link between different resources over the Web by using URI on subject, predicate and object of a RDF triple. Here URIs is glue between RDF triples and leads to a graph, [11].

One can consider a RDF triple as binary relationship with logical formula of Predicate (subject, Object). Figure 5 shows a RDF triple in graph view:

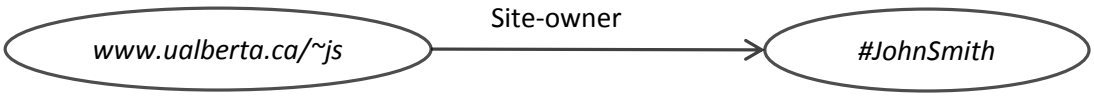


Figure 5 - RDF triple in graph view

RDF itself is a data model for building resources as RDF triples and not a data format. So for materializing it, we need a way to sterilize it. W3C has introduced two RDF serialization syntaxes, RDF/XML and RDFa. Also there are more RDF syntaxes that have been used widely, but they are non-standard syntaxes; Turtle, N-Triple and RDF/JSON, [11].

The most used format for publishing RDF statements is RDF/XML. Generally it is XML document which RDF namespace is added to top of XML document. Following shows a RDF statement in RDF/XML syntax, [11].

```
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
 xmlns:uni="http://www.ualberta.ca/uni-ns#">

      <rdf:Description rdf:about="www.ualberta.cs/~JS">
           <uni:Site-owner>JohnSmith</uni:name>
      </rdf:Description>
</rdf:RDF>
```

The element rdf:Description is used for defining a new RDF statement. The attribute rdf:about specify Subject of RDF statement by its URI. The uni:site-owner define property for www.ualberya.cs/~JS. The downside for RDF/XML syntax is that it is not suitable for human in terms of readability but it is machine-readable. Another RDF syntax is RDFa, Resource Description Framework in Attributes, which is embeds RDF statements in HTML documents. It is a set of attribute extensions to HTML and originally it was intended to add metadata. Set of attribute is: about, rel, src, href, property, datatype and typeof. Following is an example of a sample RDF statement in RDFa syntax:

```
<div about=http://biglynx.co.uk/people#dave-smith
typeof="foaf:Person">
      <span property="foaf:name">Dave Smith
</div>
```

Another way to sterilize RDF is Turtle which a plain text and compact format. It is possible to add namespaces as prefixes for RDF statements. The Turtle syntax is simply just sequences of Subject, Predicates and Object which are separated by a whitespace and terminated by '.'. Following is an example of RDF statement in Turtle syntax:

```
<http://www.ualberta.cs/~JS>
<http://www.ualberta.ca/uni-ns#Site-owner>
      "JohnSmith"
```

N-Triple is Turtle mines some features like namespace prefix, shorthand. It is clearest way to express RDF statements as each RDF triple is defined in one line. It is an advantage of N-Triple as during the loading RDF documents, each line that a triple can be parsed at a time and it is suitable for large RDF documents. Although it is clear and suitable for large datasets, but it is not shortest way as N-Triple does not support namespace prefixes so all URI should contain namespaces and it leads to larger RDF documents with compare to order RDF syntaxes even RDF/XML, [11]. Following is an example of N-Triple syntax:

```
<http://www.ualberta.cs/~JS> <http://www.ualberta.ca/uni-ns#Site-owner> "JohnSmith"
```

As we said RDF is a language for expressing resources but it does not define any semantic level to data. Also it does not have any hypothesis about domain of interest that RDF is bout. For adding semantic and describing a particular domain of interest, user can define RDFS, Resource Description Framework Schema. In the RDFS one can defines things that exist in the domain as classes and properties and relation between them. RDFS defines restriction on relation between things in the domain by domains and ranges. A class in RDFS can be considered a set of objects that each individual object is called instance of that class. The rdf:type defines relation between instances and classes in RDF and RDFS.

Another aspect of RDFs is possibility for defining hierarchy between classes and properties. In this way a class is subclass of another class if all instances of former class are instance of the later class. By this RDFS defines semantic for particular domain. The same hierarchy concept can be applied on RDFS properties. P is sub property of Q if Q(x, y) wherever P(x, y), [9].

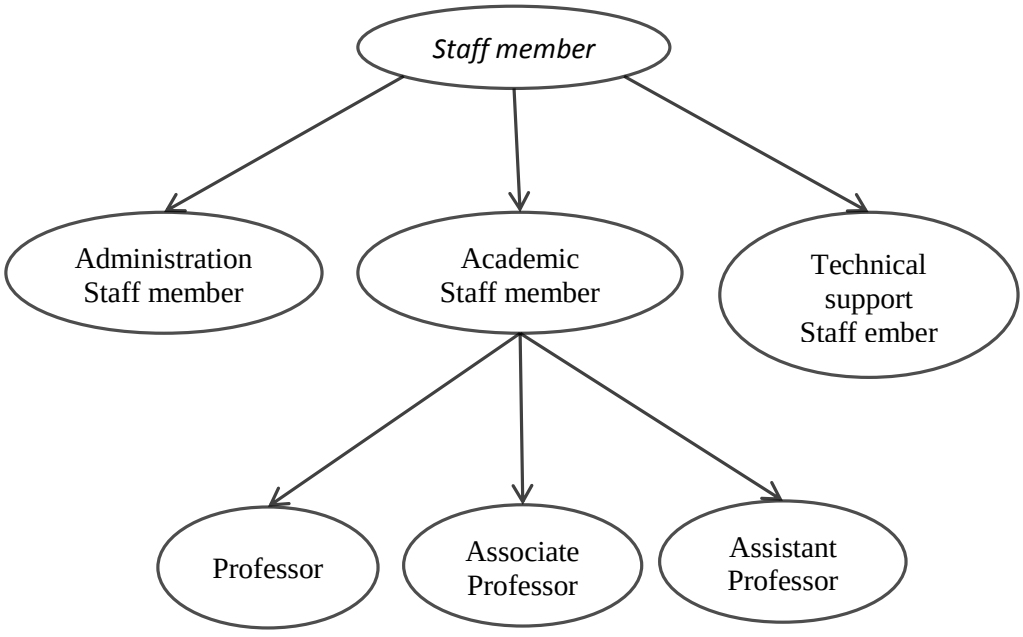


Figure 6 - RDF concepts of university resource

It is important to understand that with RDF one can define any statements about any resources and with RDFS can define restriction, subclasses and subproperties. The main classes of RDFS are: rdfs:Resource, rdfs:Class, rdfs:Property, rdfs:Literal and rdfs:Statement and main properties are: rdfs:type, rdfs:subClassOf, rdfs:subPropertyOf, rdfs:domain and rdfs:range. RDFS itself is expressed in RDF/XML syntax. Following is XML serialization of RDF concepts of university resources in Fig 6:

```
<rdf:RDF
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

      <rdfs:Class rdf:ID="lecturer">
            <rdfs:subClassOf rdf:resource="#academicStaffMember"/>
      </rdfs:Class>
      <rdfs:Class rdf:ID="academicStaffMember">
            <rdfs:subClassOf rdf:resource="#staffMember"/>
      </rdfs:Class>
      <rdfs:Class rdf:ID="staffMember">
      </rdfs:Class>
      <rdfs:Class rdf:ID="course">
      </rdfs:Class>
      <rdf:Property rdf:ID="involves">
            <rdfs:domain rdf:resource="#course"/>
            <rdfs:range rdf:resource="#lecturer"/>
      </rdf:Property>
      <rdf:Property rdf:ID="isTaughtBy">
            <rdfs:subPropertyOf rdf:resource="#involves"/>
      </rdf:Property>
      <rdf:Property rdf:ID="phone">
            <rdfs:domain rdf:resource="#staffMember"/>
            <rdfs:range rdf:resource="&rdf;Literal"/>
      </rdf:Property>
</rdf:RDF>
```

In spite of RDF is a simple and straightforward language there is some critical point of view of RDF. The binary relation is the only possible relation in RDF. In the real world we use predicates with more than two arguments. For representing these predicates we need to create some binary predicates. Also properties in RDF are considered as resource and so properties can be used as the object in a RDF triple, subject-attribute-

object. Although it makes some degree of flexibility and, but it can be confusing for modelers as it is unusual for modeling languages.

2.5. OWL

OWL [23], Web Ontology Language; also it defines relations and instances of classes. In order for making the Web understandable by machines and software agents and performing useful reasoning tasks on the web, it is necessary to go beyond keywords and give meaning to defined resources and concepts over the Web.

The OWL language has three variant sublanguages with different level of expressiveness. OWL Lite, OWL DL and OWL Full are these sublanguages. Each sublanguage is an extension of its predecessor. The following are true relations between these three OWL sublanguages:

- Every legal OWL Lite ontology is a legal OWL DL ontology
- Every legal OWL DL ontology is a legal OWL Full ontology
- Every valid OWL Lite conclusion is a valid OWL DL conclusion
- Every valid OWL DL conclusion is a valid OWL Full conclusion

OWL Lite is intended for defining classification hierarchy and simple constrains. It uses just some of OWL language features and has more restriction and limitation than OWL DL and OWL Full. For example OWL Lite has a limited cardinality values as 0 or 1. Another example of restriction in OWL Lite is equivalence of classes and subclasses relationships between classes are just allowed between named classes.

OWL DL guaranteed that all entailments will be computed, completeness, and computations will finish in finite time, decidability. While OWL DL supports completeness and decidability, it has maximum expressiveness. It is called OWL DL because of its correspondence with description logics.

OWL Full has maximum expressiveness without computational guarantees and syntactic freedom of RDF. In an ontology with OWL Full syntax, someone can add the

meaning of the predefined, either RDF or OWL, vocabulary. There is not much reasoning software to support all features of OWL Full.

An Ontology language should allow users to write explicit formal conceptualization of domain models. Following are main requirements for an ontology language:

- A well-defined syntax
- Efficient reasoning support
- A formal semantic
- Sufficient expressive power
- Convenience of expression

Two requirement of an ontology language are important: expressive power and efficient reasoning support. The richer the language is, the more inefficient the reasoning support becomes. There is always a compromise between expressive power and efficient reasoning support. The most useful ontology language is the one supported by reasonably efficient reasoners and the one that can express large classes of ontology and knowledge.

2.6. Time / Temporal OWL / Temporal RDF

Time is one of the most common concepts that could be found in spoken languages, systems and applications. Many real-world applications require management of temporal data. Because of importance of time in applications and domain knowledge, some custom temporal management solutions were developed. In particular, biomedical data, that time dimension is central, forced research in the area, [32]. One of the first biomedical systems to address the problem was the Time Oriented Database.

In the context of Web of data sharing, accessing and using temporal data is necessary and current technologies are either complex or not sufficient for handling temporal thing in Web of Data, [33].

OWL and SWRL are powerful technologies but they have limitation to handle temporal information. OWL just supports temporal data values as basic XML Schema dates, times and durations, [21]. SWRL includes operators for manipulating these values at a very low level. There are two approached to add time dimension. First of all, add temporal dimension to OWL that is not straightforward. Another way is developing a time model on top of OWL. In the second approach, we can model time at the user level without changing OWL itself. Our Time ontology model has three main subclasses, ValidTime, Event and Granularity. ValidTime just concerns about two main concepts of time, instant and interval that is representative of a moment and extend on time line. Topology of instant and interval and possible relations between two proper intervals can be viewed in figure 7 and figure 8, [30].

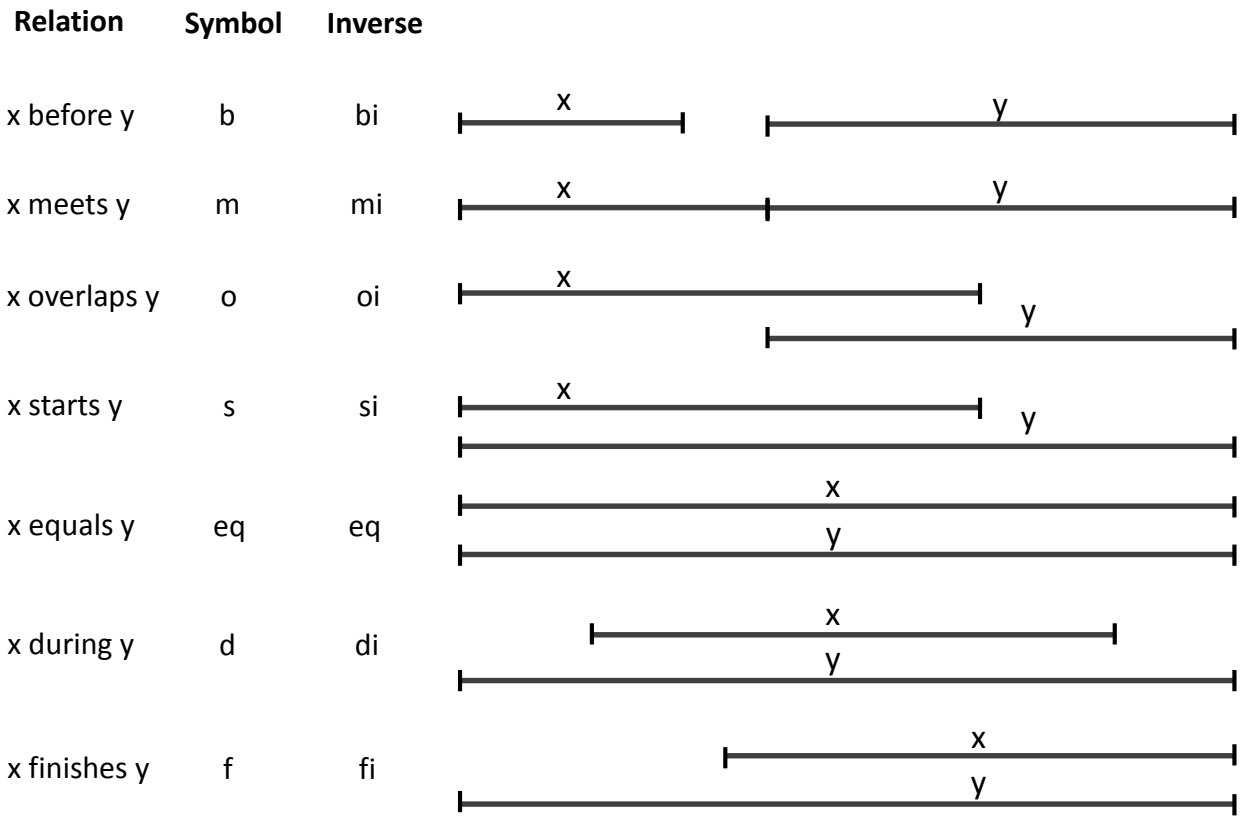


Figure 7 - Allen's interval relations

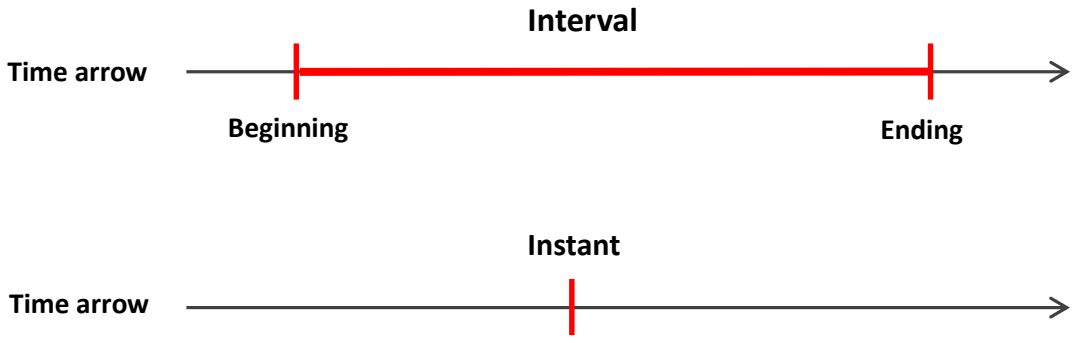


Figure 8 - Instant and Interval in time line

The approach described in [29] represents temporal knowledge in an interval-based temporal logic. Such a model is suitable for expressing qualitative temporal relation between intervals such as before, after, meets and overlaps. Dutta [39] has proposed a model that describes time using a set of accurate disjoint intervals that can have different lengths, while Kim and Oh [10] have introduced a model for the vague representation of qualitative temporal relations between events.

Pan and Hobbs [8] have introduced a model for representing time in an OWL-represented ontology [23]. They offer a comprehensive description of temporal intervals, instants, durations, and calendar terms. They provide a vocabulary to represent facts about topological relations among instants and intervals.

In general, introduction of time into RDF has been accomplished via creating a new extended version of RDF. For example, in [6][5] the authors use labeling and special data structures to introduce time into RDF graphs. However, the proposed query language is almost impossible to use in existing RDF stores. In the attempt to solve the storage problem, the authors of [3] introduce the tGRIN index structure that builds a specialized index for temporal RDF that is physically stored in a rational database. They use temporally annotated RDF triples of the form: subject-property:annotation-object. A similar approach – annotated triples – is used in [28]. Here, the authors focus on temporal validity intervals. They propose a temporal version of the SPARQL language for querying such annotated triples.

As we mentioned earlier, RDF is recommended metadata model and language by W3C for building Web of Data. RDF is constructed by RDF triple that is a binary relation between subject and object via a predicate. Binary relation is simple and powerful relation not only for building huge graph of triples but also for building a querying language among RDF data. This binary relation has restriction.

2.7. Fuzzy Logic

The concept of Fuzzy theory and set has introduced by Zadeh. In general fuzziness means uncertainty boundaries among set of objects. For instance, if someone wants assign a people to one of the human height's concept like "Tall" or "Short" in dichotomy approach, a certain threshold is needed to be defined for constructing concepts over human height's discourse, Fig 9[1].

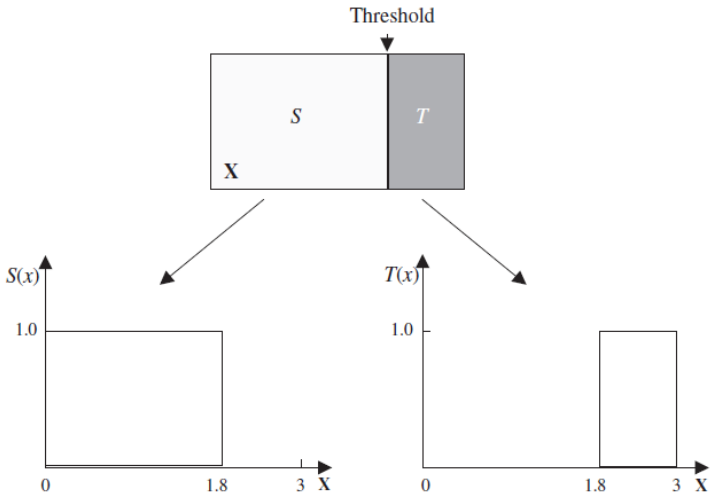


Figure 9 - Dichotomy sets and their characteristic functions

In the other hand, in fuzzy logic each example can be assign to each concept with a degree of membership, Fig 10.

[1] All fuzzy membership function figures have been borrowed from [46]

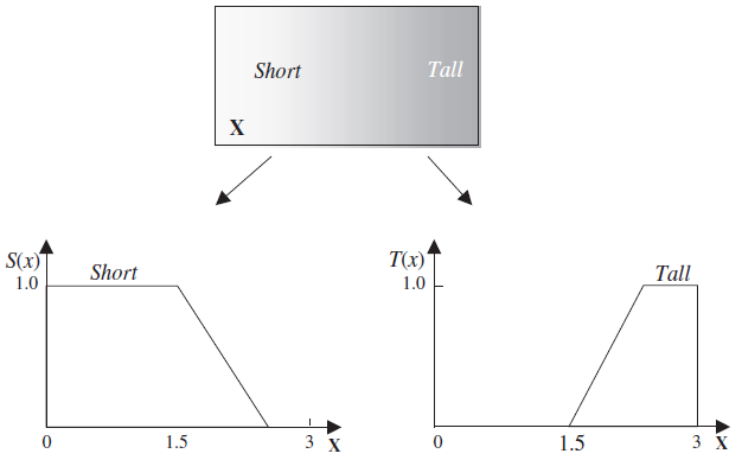


Figure 10 - Fuzzy sets and their membership functions

Fuzzy membership function is definition of each concept over the universe of discourse. Someone can consider Fuzzy sets as a set of pairs of the {x, A(x)} which x is a sample of X and A(x) is its degree of membership. Here A(x) explains the degree of compatibility of example x to concept A. This is the main explanation of fuzzy sets, [21].

Generally any function A: X -> [0,1] can be considered a fuzzy membership function, but it should reflects the purpose of constructing fuzzy set. The fuzzy membership functions should mirror the level of detail we intend to capture and the perception of the concept to be represented and used in problem solving, [46].

Following are common used categories of fuzzy membership functions:

- **Triangular Membership Functions**

The pairwise linear segments describes triangular membership function with a, m and b parameters.

$$A(x,a,m,b) = \begin{cases} 0, & \text{if } x \le a \\ \dfrac{x-a}{m-a}, & \text{if } x \in [a,m) \\ \dfrac{b-x}{b-m}, & \text{if } x \in [m,b] \\ 0, & \text{if } x \ge b \end{cases}$$

The above expression can be written in the form of,

$$A(x, a, m, b) = \max\{\min[(x-a)/(m-a), (b-x)/(b-m)], 0\}$$

Which m denotes a modal value of fuzzy set whereas “a” and “b” are the lower and upper bounds, Fig 11.

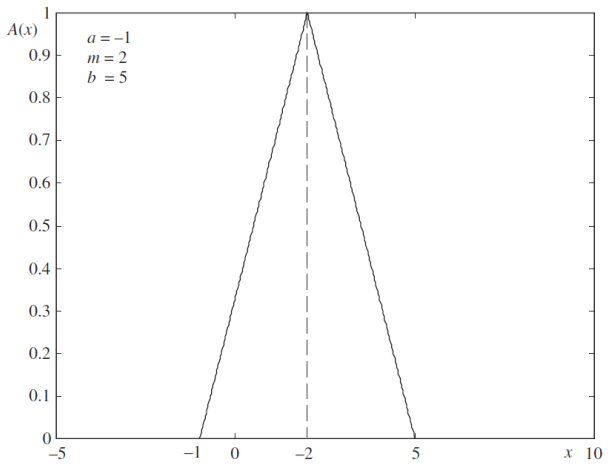


Figure 11 - Triangle membership function

- **Trapezoidal Membership Functions**

The pairwise linear segments describes triangular membership function with a, m, n and b parameters.

$$A(x) = \begin{cases} 0, & \text{if } x < a \\ \dfrac{x-a}{m-a}, & \text{if } x \in [a, m) \\ 1, & \text{if } x \in [m, n) \\ \dfrac{b-x}{b-n}, & \text{if } x \in [n, b] \\ 0, & \text{if } x > b \end{cases}$$

The above expression can be written in the form of,

$$A(x, a, m, n, b) = \max\{\min[(x-a\}/(m-a),\ 1,\ (b-x)/(b-n)], 0\}$$

Figure 12 illustrates the Trapezoidal Membership Function.

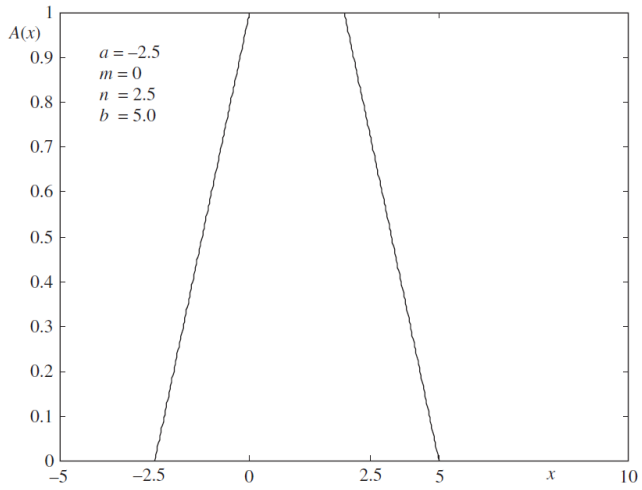


Figure 12 - Trapezoidal membership function

- **Γ-Membership Functions**

They are expressed in the form of:

$$A(x) = \begin{cases} 0, & \text{if } x \le a \\ 1 - e^{-k(x-a)^2}, & \text{if } x > a \end{cases} \quad \text{or} \quad A(x) = \begin{cases} 0, & \text{if } x \le a \\ \dfrac{k(x-a)^2}{1 + k(x-a)^2}, & \text{if } x > a \end{cases}$$

Where k > 0, as illustrated in figure 13:

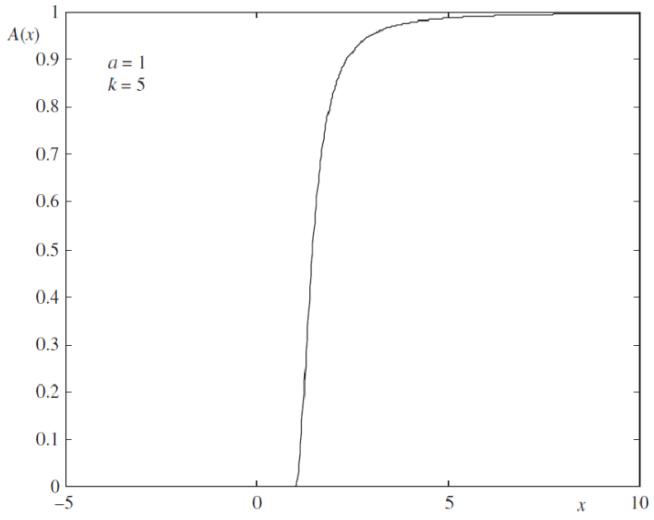


Figure 13 - Γ-Membership Function

- **S-Membership Functions**

They are expressed in the form of:

$$A(x) = \begin{cases} 0, & \text{if } x \le a \\ 2\left(\dfrac{x-a}{b-a}\right)^2, & \text{if } x \in [a, m) \\ 1 - 2\left(\dfrac{x-b}{b-a}\right)^2, & \text{if } x \in [m, b] \\ 1, & \text{if } x > b \end{cases}$$

Figure 14 shows its memebrship function:

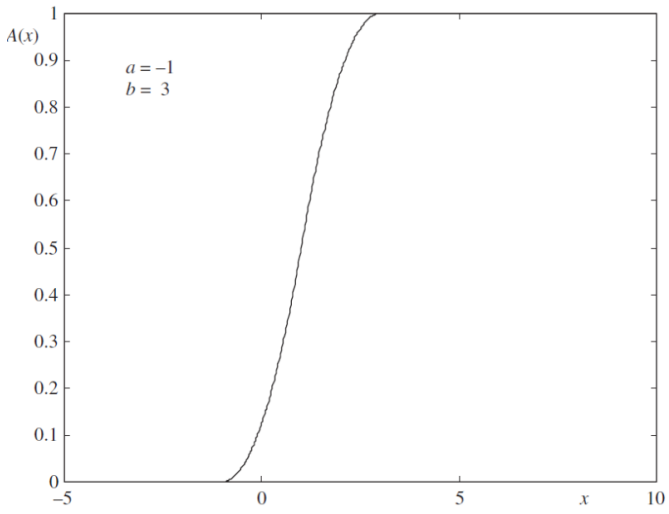


Figure 14 - S-Membership Function

- **Gaussian Membership Functions**

The following shows these membership functions:

$$A(x, m, \sigma) = \exp\left(-\frac{(x-m)^2}{\sigma^2}\right)$$

In the Gaussian membership function the modal value m denotes the typical element of A and σ represents a spread of A, Fig. 15.

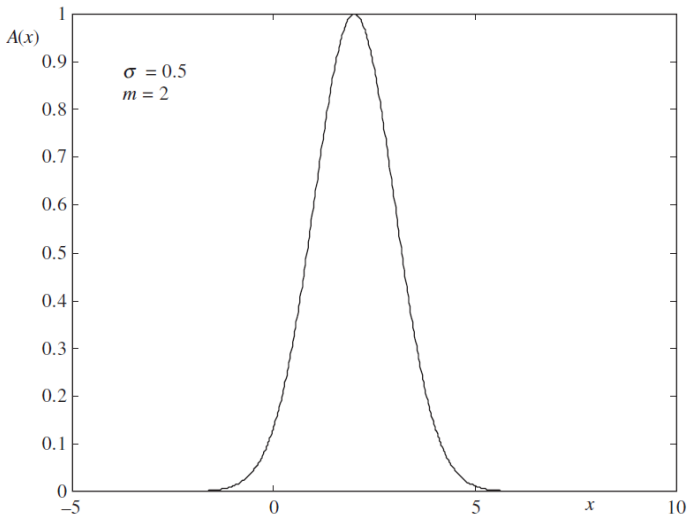


Figure 15 - Gaussian Membership Function

In practice, absence of knowledge and incomplete information on system inputs, parameters and structure lead to uncertainly and imprecision, [46].

2.8. Vague Temporal and Fuzzy Logic

There are studies that focused on crisp temporal concept in ontology. But, in real world many of the temporal concepts are vague and imprecise, and it is not easy dealing with them. Fuzzy theory is a valid solution for handling these imprecise temporal concepts.

Dubois and Prade [7] have proposed an approach to represent and process fuzzy temporal knowledge. Carinena at el [36] have focused on providing a formal definition of a grammar for expressing fuzzy temporal propositions. The definitions of such concepts as date, time extent, and interval, according to the formalism of possibility theory have been presented in [37].

Lamine and Kabanza introduced a set of fuzzy temporal operators that is corresponded with a set of classic temporal operators. They used operations in Zadeh's interpretation instead of Boolean connectives such as and, or, not. Another approach was

proposed by Dubois and Prade [7] to represent and to process of fuzzy temporal knowledge, [7]. They introduced four main concepts: (i) time as a one-dimensional discrete axis, (ii) date as an instant of time, (iii) time extent that represents quantity of time, and (iv) interval as a period of time.

Carinena at el [36] have focused on providing a formal definition of a grammar for expressing what they called fuzzy temporal propositions. They describe the language and show its application for determining degree of fulfillment of the proposed temporal propositions. The definitions of the concepts of date, time extent, and interval, according to the formalism of possibility theory have been presented in [37]. The authors introduce relations between the temporal entities such as dates and intervals interpreted as constraints on the distance between dates, and projected onto Fuzzy Temporal Constraint Satisfaction Networks.

Another proposed work by Dutta is defining possibility of occurrence of an event in a time interval, [39]. One can evaluate temporal relation between a pair of events.

2.9. Related Work

Introducing time into relational databases was studied for adding time dimension. In particular, biomedical data, that time dimension is central, forced research in the area [32]. One of the first biomedical systems to address the problem was the Time Oriented Database. Because of importance of time in applications and domain knowledge, modeling time in Web of Data and particularly in RDF modeling is one of the key primitives. RDF itself just supports primitive date time that comes from XML date time datatype. The first approach for modeling temporal information in RDF was introduced by Gutierrez and Hurtado. They introduced determinate RDF triples and a query language.

Generally there are two main approaches for adding time to RDF, versioning and time labeling. In the versioning approach, each change in triples of a RDF graph leads to creating a new version of RDF graph and saving old RDF graph. In the second approach,

labeling, for each change on RDF triple a new label with date time is assign to changed element. Valid time and transaction time are two different dimensions for representing temporal relations in temporal databases. The first one represents the time that data is valid in the model and the second one is the time that data is saved in the database. Generally labeling approach is more suitable for RDF modeling as in case of large scale RDF model and when changes are frequent versioning approach may have large overhead and performance issues on querying data. Also labeling approach follows nature of RDF in extensibility and distributing, [5].

Pugliese and Udrea extended Gutierrez work for case of indeterminate RDF triples. Also they introduced normalized tRDF database and normalization algorithm. The most used representing time in temporal database is a discrete and ordered linear representation and also it is used mostly for time in modeling time in RDF. In this representation, two main concepts are instant and interval that represent a moment and extend on time line. An interval time is a period between two instant of time. Topology of instant and interval and possible relations between two proper intervals has been introduced [30].

In general, some researchers focused on introducing concept of time into RDF by creating new extended RDF, querying language and temporal RDF repository for handling time like [6], [3], [1], [47] and [10]. Another approach is introducing time to RDF by using standard RDF and annotation for temporal dimension. In this approach time is encoded in the user data model; in this case handling temporal dimension for querying data is necessary. [32] used SWRL rules for retrieving data.

The authors of [38] have focused on extending an existing framework of representing temporal information into ontology. They have proposed representation of concepts evolving in time, dynamic concepts, together with their properties as qualitative descriptions. They used natural language expression for temporal events without exact duration, starting or ending points. For the purpose of handling dynamic concepts they used two different mechanisms: 4D-fluents and N-ary relations. They have also used two concepts for expressing temporal information: interval and instant.

Methodology of handling temporal information containing uncertainty has been addressed in [42]. The authors have proposed modeling time using two different models: linear model and branching model. The first of the models is called Fuzzy Linear Temporal Logic or FLTL [41], while the second model proposed here is Fuzzy Branching Temporal Logic or FBTL. In the branching model, an event can have a number of possible branches. They used fuzzy logic for handling uncertainty in the logical information, and fuzzy temporal primitives for handling temporal information. They used three different types of temporal primitives: time interval, interval with possibility measures, and a fuzzy representation of a time instant.

Hobbs and Pan [8] have introduced a model for representing time in an OWL-represented ontology. They offer a comprehensive description of temporal intervals, instants, durations, and calendar terms. They provide a vocabulary to represent facts about topological relations among instants and intervals. The authors propose the class of eventualities to "cover events, states, processes, propositions, states of affairs, and anything else that can be located with respect to time" [8]. In the proposed model [8], the term "duration description" is different from the duration concept, i.e., the duration of an interval can have many different duration descriptions. It is useful to talk about descriptions as independent objects. They also introduce clock and calendar to distinct intervals. For instance, they indicate that a day as duration and a day as calendar interval are different. Additionally, Hobbs and Pan argue that time zone should be considered in the concept of time. It is to be mentioned that all parts of date time except seconds are related to time zone. They develop a time zone resource in OWL, which defines the vocabulary about regions, time zones, daylight saving policies, and the relationships between them.

The main goal of the paper [31] is to integrate uncertainty with Allen's interval-based temporal logic. The authors have defined new formalism for extending classical interval algebra, as well as developed a temporal reasoning system capable of handling both qualitative and quantitative information. Their approach applies Temporal Constrain

Satisfaction Problem for building queries for temporal occurrence and relation between them.

The issue of collecting information hampered by vagueness and uncertainty that comes from unreliable data source has been addressed in [2]. The authors have focused on uncertainty of the information. They have proposed a temporal framework, called Fuzzy-time Temporal Logic or FTL that takes into account a degree of truth of temporal expressions. A set of fuzzy temporal modalities has been defined in the context of this framework that respects a set of expected mutual relations. The framework is usable for crisp events and in this situation their FTL framework reduces to LTL.

The issue of reasoning with fuzzy temporal data has been touch upon in [48] and [40]. The first of the papers has proposed an extension to an existing temporal reasoning framework. It allows for managing uncertainty based on a many-valued logic. An extended reasoning algorithm has been provided. It can handle both temporal and uncertain information in an integrated way. The authors have used many-valued logic by means of the Lukasiewicz logic. In [40], the authors have focused on structured temporal information, and the lack of precise boundaries of historical events. They have proposed a framework based on fuzzification of Allen's Interval Algebra. They applied it to information retrieval from Web documents.

3. Fuzzy Temporal Data in Ontology Environment

The concept of Semantic Web has introduced an important form of knowledge representation – ontology. As a hierarchical structure of concepts together with their definitions, ontology provides means for expressing semantics of data. The ability to build rules with ontology concepts and to perform reasoning increases its attractiveness even further. This chapter describes a framework for expressing fuzzy temporal information using ontology. The framework is built based on ontology suitable for expressing facts and building rules that include fuzzy and temporal terms. This proposed fuzzy temporal ontology can be imported to any domain ontology and used a knowledge base in variety of applications. The chapter includes description of build-in predicates needed for constructing fuzzy temporal rules. Simple examples of application of the predicates are presented.

3.1. Ontology For Representing FUZZY and Temporal Data

A. Ontology of Fuzzy Data

The fuzzy ontology used in the proposed framework should be capable of expressing fuzzy information. Therefore, the ontology we propose contains a number of concepts that are required for representing and storing fuzzy information. Its description can be divided into two parts: an overview at the definition level explaining classes and properties necessary to define fuzzy variables; and an overview at the individual level illustrating an application of fuzzy ontology for representing concrete fuzzy information.

1) Definition Level Description: The hierarchy of classes of the ontology for representing fuzzy information is shown in Fig 16.

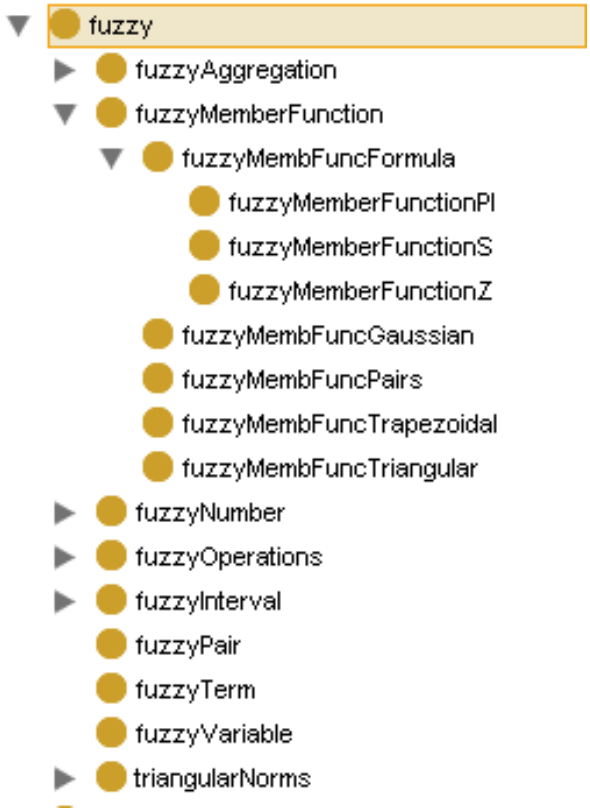


Figure 16 - Ontology for representing fuzzy information

It contains a set of concepts from the domain of fuzzy sets and systems. The concepts denote basic items required to express fuzzy data. The most important concepts are:

- *fuzzyVariable* defines a class of fuzzy variables of interest, the definition includes datatype properties required to define a fuzzy variable – universe of discourse: *discourseMin*, *discourseMax*; name: *fVarName*, and a set of associated terms: *fTermSet*;

- *fuzzyTerm* is a class of entities representing fuzzy linguistic labels defined for a given fuzzy variable;

- *fuzzyPair* defines a class of pairs <membership value/element of discourse>;

- *fuzzyMemberFunction* is a class of membership functions associated with fuzzy terms, as we can see, Fig. 16, there are a number of subclasses representing different types of membership functions.

An important part of the ontology – necessary for "building" fuzzy information – is object properties. A list of properties is shown in Fig. 17. The properties define fundamental relations that exist between different fuzzy concepts.

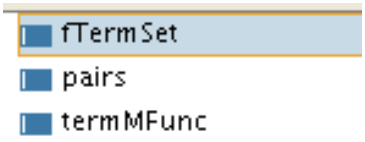


Figure 17 - Object properties

The arrangement of both classes and object relations allows us to build a structure needed for expressing fuzzy information. Such a structure is illustrated in Fig. 18. The *is_a* relations (representing hierarchical structure of the ontology, Fig. 16) are not so essential when compared with the object relations. The relations between classes reflect "real-world" process of constructing fuzzy variables, i.e., each *fuzzyVariable* is composed (connected via the relation *fuzzyTerms*) with multiple linguistic labels (*fuzzyTerms*) defined in the universe of discourse of the variable, and further each label/term is linked with a membership function – it is done via the relation *termMFunc*. Membership functions could be of any different type: Gaussian, trapezoidal, and so on.

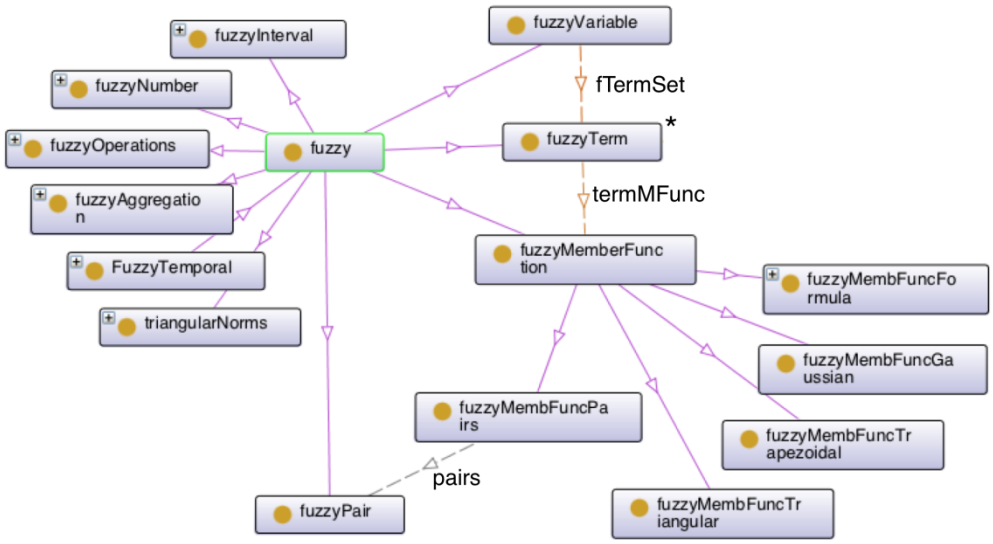


Figure 18 - Ontology classes and properties as a structure for defining fuzzy information (* indicates that multiple fuzzy terms can be associated with a single fuzzy variable.)

2) Individual Level Description. A better understanding how to apply the proposed fuzzy ontology for representing fuzzy data can be achieved with an example of representing a concrete piece of information. This example shows an ontology-based definition of a fuzzy variable named *BoilerTemp* that is an instance of the class *temperature*. The variable contains three fuzzy labels/terms *low*, *medium*, and *high* linked with three membership functions. The definition of this variable is revealed in Fig. 19. Let us emphasis the fact that concrete pieces of information are represented as individuals (marked with diamonds) that is instances of specific/required classes (marked with circles) connected via proper object relations.

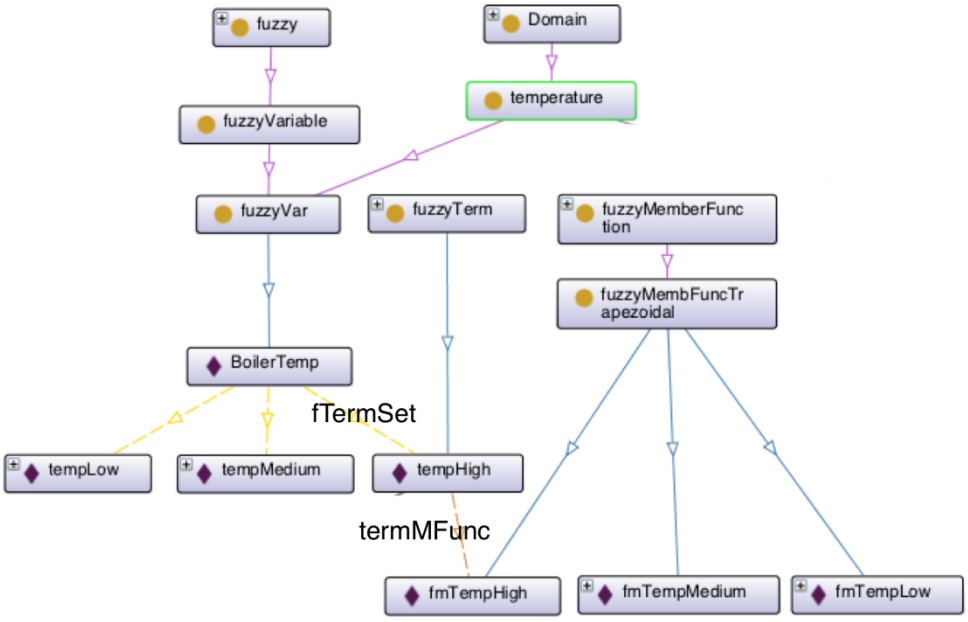


Figure 19 - Ontology-based definition of a fuzzy variable temperature

As it can be seen, the variable *BoilerTemp* is an instance of the class *fuzzyVar* and it inherits features of two classes *fuzzyVariable* (presented above) and *temperature*. The *BoilerTemp* is linked via the property *fTermSet* with three labels: *tempLow*, *tempMedium*, and *tempHigh*. The labels are associated, via property *termMFunc*, with membership functions. Fig. 19 shows this only for the label *tempHigh*. Please note that membership functions are instances of another class: *fuzzyMembFuncTrapezoidal*, which is a subclass of *fuzzyMemberFunction*.

B. Temporal Ontology for Fuzzy Data

In order to express temporal fuzzy data we define a simple ontology of basic temporal concepts based on the time ontology described in Section II.C. In the ontology, time is represented as a single time point, or as a time interval. A time point, or *time instant*, identifies a single occurrence, while a *time interval* is a temporal entity with a beginning time and an ending time. Usually, the beginning and ending are determined with time instants. An important aspect of temporal data is time stamping of information.

This is done with so-called *Valid Time*, i.e., the time that information is true in the real world.

The temporal ontology, or should we say ontology for temporal fuzzy data, is shown in Fig. 20. The defined classes, based on terms presented in Section II.B, allow for representing basic temporal concepts. The concepts are:

- *ValidTime* is a class representing possible "moments" when information is true in the real world. It includes two object properties: *activation* that links it with the class *fuzzyTerm*, and *hasGranularity* linking it with the class *Granularity*. It is a superclass for two other classes: *Instant* and *Interval*. The class *Instant* has a datatype property *hasTime*, while the class *Interval* has two datatype properties *hasStartTime* and *hasFinishTime*.

- *Event* defines a class of entities occurring in time, it has two subclasses *InstantEvent* and *IntervalEvent* that are also subclasses of the classes *Instant* and *Interval*, respectively. The fact that *InstantEvent* (*IntervalEvent*) is also a subclass of *Instant* (*Interval*) allows it to inherit properties required to define a temporal event and link it with a fuzzy term;

- *Granularity* is a simple class used to identify time granularity of individuals of the class *ValidTime* (and via inheritance of *InstantEvent* and *IntervalEvent*), its individuals are different time units, for example *Seconds*, *Minutes*, *Hours*, and so on.

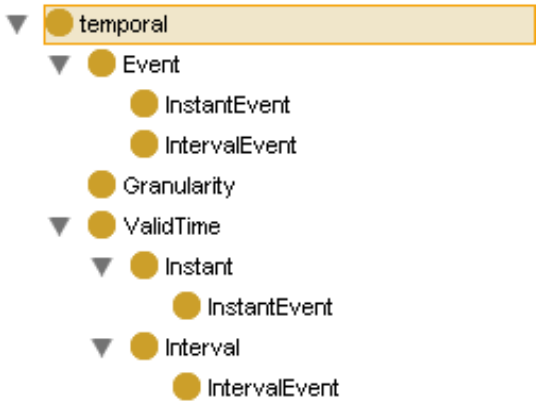


Figure 20 - Ontology – classes and properties – for temporal data

We combine this ontology with the ontology for fuzzy data (previous subsection). A fuzzy variable class *temperature* presented in the previous subsection can be augmented with a temporal aspect, i.e., it becomes a superclass of the class *dataTemporal* that is also a subclass of the class *instantEvent*, Fig. 21. An instant of the class *dataTemporal* is named *dataTemporal_1*. It is linked with the instance *Hours* of the class *Granularity* via the property *hasGranularity*.

The representation of a series of *BoilerTemp* measurements is done by generating a set of instances *dataTemporal_x*. Each of them contains a time stamp, a value, and is linked with granularity (*Hours* in our case) and an instance of *fuzzyTerm*.

3.2. FUZZY Temporal Predicates

In order to deal with temporal data, it means, to construct rules that take into account temporal relations and temporal dependencies between different events, we need a library of constructs capable of dealing with temporal instances and intervals. Such a library gives us the ability to build rules that can express relations between temporal events and infer about them. For example, the rule

```
if
   signal S1 is high about 10 sec before signal S2 is low
then
   value of A should be 5
```

has a component – *about 10 seconds before* – that requires a special temporal predicate for expressing time interval or instance in an approximated manner. A number of fuzzy temporal predicates have been identified and implemented in Protégé (ontology editor: http://protege.stanford.edu/).

Before we describe the predicates, let us define basic terms that are used in the implemented temporal predicates:

```
timeInstant = <NOW|…|specificDateTime|timeOfAnotherEvent>
```

used for expressing instant in time, it could be given explicitly with such words as NOW, YESTERDAY, LAST_SECOND, LAST_MINUTE, LAST_HOUR, LAST_DAY, or a specific date and time, for example "2012-05-06 16:23", or implicitly via specifying a temporal event (for example, *dataTemporal_1*, Section III.B), the time stamp of this event is used to determine a specific instant in time;

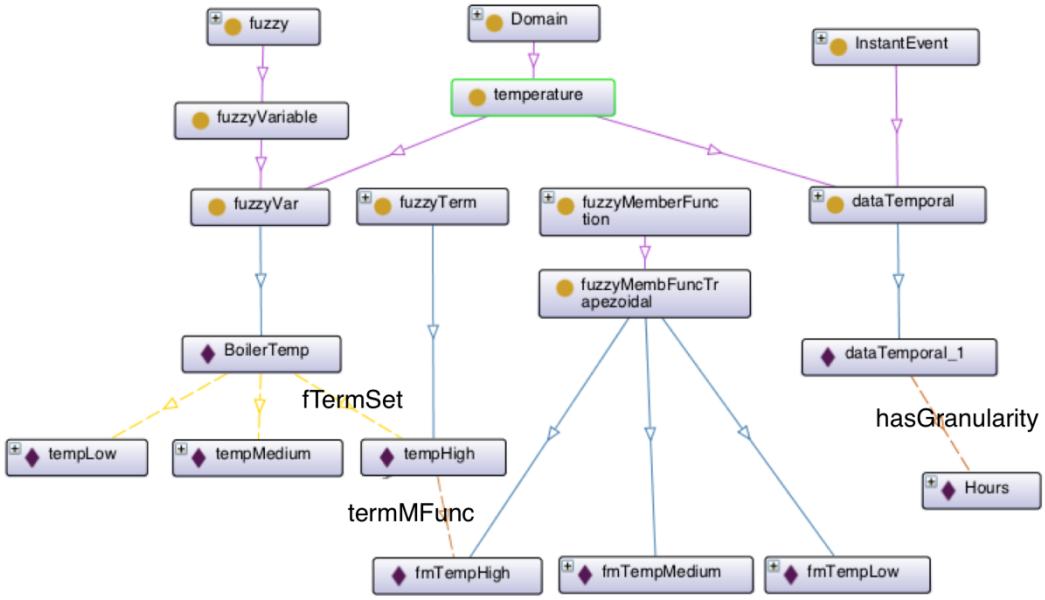


Figure 21 - Fuzzy temporal variable temperature

```
timeInterval = <(periodStartDate, periodEndDate)>
periodStartDate|periodEndDate =
<specificDateTime|timeOfAnotherEvent>
```

for defining a time interval via providing the beginning and ending time instants – explicitly via a specific date/time, or implicitly via events.

A. Fuzzy Temporal Predicates

Predicate #1:

approx_at_instant(?degree1, ?event, ?timeInstant)

This predicate is used to determine the degree of temporal overlapping between an event and an approximated instant in time. In other words it calculates the level of degree to which 1 In SWRL, the character "?" is used to denote a variable. The statement "something happened approximately around some time instant" is true. The parameters are:

degree – output value from the interval <0,1>, it is a degree to which the *event* overlaps with "about" *timeInstant*;

event – input, usually an individual from ontology with a value and a time stamp, we are determining if this *event* happened "about" *timeInstant*;

timeInstant – input, it is a moment in time defined in a number of ways (see above).

Predicate #2:

approx_in_interval(?degree, ?event, ?timeInterval)

This predicate is useful to determine the degree of overlapping between an event and an interval defined by starting and ending points. Its parameters are:

degree – output value from the interval <0,1>, it is a degree to which the *event* overlaps with the interval;

event – input, usually an individual from ontology with a value and a time stamp, we are determining if this *event* happened in the interval;

timeInterval – input, defined as above.

Predicate #3

approx_at_instant_before(?degree, ?event, n, granularity, ?referenceInstant)

This predicate is to determine the degree of overlapping between an event and an instant that occurs about n granules of time before some instant of time. For example, the predicate *approx_at_instant_before(?degree, ?event, 2, days, ?referenceInstant)* will

calculate the degree of overlapping of an *event* with an instant at about 2 days before ?*referenceInstant*. Its parameters are:

degree – output value from the interval <0,1>, it is a degree to which the *event* overlaps with the moment that happened n time granules before *referenceInstant*;

event – input, usually an individual from ontology with a value and a time stamp, we are determining if this *event* happened at about n time granules before *referenceInstant*;

n – number of time units;

granularity – determines the units of *n*;

referenceInstant – input, defines a moment in time, it is defined using *timeInstance* (see above).

Predicate #4

approx_in_interval_before(?degree, ?event, n, nGranularity, ?referenceInstant, m, mGranularity)

This predicate determines the degree of overlapping of an event with an interval which spans across *n* granules of time, and this interval occurs *m* granules of time before *?referenceInstant*. The parameters are:

degree – output value from the interval <0,1>, it is a degree to which the *event* overlaps with an interval that spans over *m* granules and occurs *n* granules before *referenceInstant*;

event – input, usually an individual from ontology with a value and a time stamp;

n – input, number of time units, "width" of an interval;

nGranularity – input, determines the units of *n*;

referenceInstant – input, it is a moment in time defined as *timeInstant* (see above);

m – input, number of time units "between" the end of the interval and *referenceInstant*;

mGranularity – input, determines the units of *m*.

Additionally, there is a need for a predicate that determines a degree of belonging of an even that has a value (in a specific domain) in a fuzzy term (linguistic label). This predicate is:

```
fuzzification(?event, fuzzyTerm)
```

Where the parameters are:

event – input, an individual from ontology with a value;

fuzzyTerm – input, an individual of the concept fuzzyTerm (has membership function and label).

This predicate does not return the value of membership, it returns true if the membership is nonzero, and false if it is zero.

B. Implementation Remark

The presented predicates are implemented as SWRL built-ins. They can be used to build rules with fuzzy temporal components in the environment of Protégé 3.4. The reasoning process is preformed using Jess (crisp part) and FuzzyJ (fuzzy part). The development has been done using Protégé and FuzzyJ APIs in Eclipse.

3.3. Illustrative Example

A. Rule with the Predicate Instant Before

The first example illustrates the application of the predicate #3. The "natural language" form of the rule is presented below:

```
if
    signal S1 is high at about 5 sec before signal S2 is low
then
    …
```

We consider only activation of the rule's antecedent without restricting the format of the consequence. In SWRL, the rule is:

```
S1(?x) ∧ fuzzification(?x, high)∧
S2(?e) ∧ fuzzification(?e, low)∧
approx_at_instant_before(?o, x, 5, seconds, ?e)
->...
```

The components of the rule are interpreted in the following way. The first term *S1(?x)* "returns" all individuals that are instances of the signal defined as the concept S1. Each individual has two parameters: value, and a time stamp. The second component *fuzzification(?x, high)* "takes" these individuals and based on their values determines their membership value in the fuzzy term identified by the linguistic term *high*. Again, the S2(?e) returns individuals of the signal S2, and *fuzzification(?e, low)* determines their membership values in the fuzzy term *low*. At this point, all individuals of the signal S1 that have a non-zero membership value in the fuzzy term *high*, and all individuals of the signal S2 that have a nonzero membership value in the term *low* are identified. Now, all these individuals are processed by the built-in predicate *approx_at_instant_before*. This predicate "checks", based on the time stamps, if an individual of the signal S1 is in the state *high* at about 5 seconds before an individual of the signal S2 is in the state *low*. The resulting waveform is shown in Figure 22.

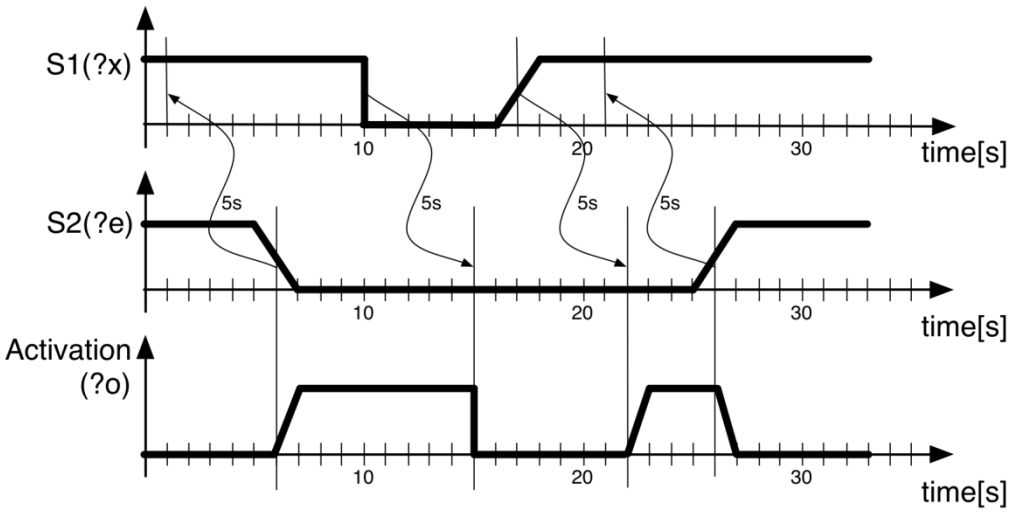


Figure 22 - Activation of rule's antecedent

B. Rule with Multiple Predicates

For the second example we have selected a more realistic rule related to weather and road conditions. The rule determines the circumstances of developing slippery roads. One of possible rules provided by an expert

```
if
     (snowfall is heavy and temp is low for about 5 hours)
     about 2 hours before
     (temp is very low for about 6 hours)
then
     slippery
```

indicates that roads will be slippery if a very low temperature is preceded by a heavy snowfall and a low (around zero) temperature. The temporal illustration of the situation leading to the slippery roads is presented in Fig. 23. The SWRL rule representing the expert's knowledge about slippery roads is shown below.

```
snowfall(?s) ∧ fuzzification(?s, high)∧
approx_in_interval_before(?o, ?s, 5, hours, NOW,8, hours)∧
temp(?t1) ∧ fuzzification(?t1, aroundZero)∧
approx_in_interval_before(?o,?t1, 5, hours, NOW,8, hours)∧
temp(?t1)∧ fuzzification(?t2, veryLow)∧
approx_at_instance_before(?o,?t2, 6, hours, NOW)
->slippery
```

The rule's antecedent can be "divided" into three parts: the first two (about *s* and *t1*) are used to determine conditions preceding a very low temperature. The *s* represents hourly snowfall: it is checked if snowfall is *high* and if it occurred over a period of 5 hours 8 hours ago. The *t1* that stands for temperature is checked if it is *aroundZero* for the same temporal interval. The third part is used to determine if the temperature is *veryLow* in the last 6 hours.

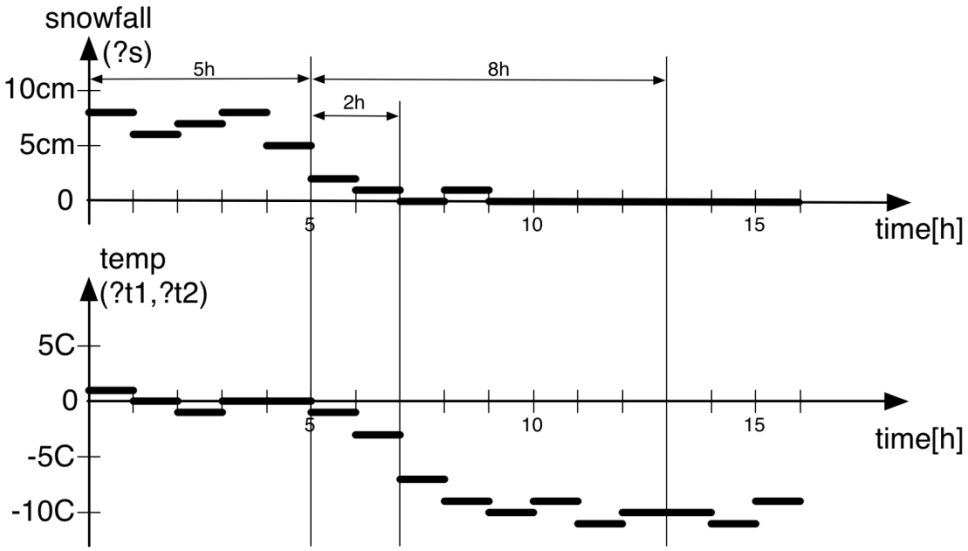


Figure 23 - Conditions leading to slippery roads

4. LORI: Linguistically-Oriented RDF Interface For Querying Fuzzy Temporal Data

The concept of Semantic Web, introduced by Berners-Lee in 2001, emphasizes importance of expressing semantics of data stored on the web. The introduced data format called Resource Description Framework (RDF) is a meaningful way of expressing and exploring data relations. It provides basics for constructing semantics-oriented data formats. More and more often RDF is used to represent variety of data including N-ary relations and temporal information. On many occasions this results in complex data structures. Their utilization requires a full understating of used data configurations.

This chapter presents a fuzzy-based Linguistically Oriented RDF Interface – LORI – for querying RDF data containing temporal information and built using non-trivial data structures. The interface includes specialized build-in predicates suitable for constructing temporal queries and supporting imprecise phrases describing time and data features, and high-level predicates built based on them. A simple case study using LORI interface for imprecise querying about time-based events is presented.

4.1.RDF and Time Representation

4.1.1. RDF Representation of N-ary Relations

In its canonical form, RDF is used to represent a binary relation. An RDF predicate additional information that is related to a particular RDF predicate. A solution supported by W3C, based on N-ary Relation [26], is to represent a relation as a class. It means that any predicate that needs to be described by additional features, e.g., strength, certainty, is

an item "located" in the middle of the original triple. The idea is presented below. Let us assume that John traveled to Tokyo. We express this in the following way:

John **travel** Tokyo

If we want to provide additional information about the relation travel such as: duration, date, stayed-at, we need to create a class **travelC** and a number of properties: travel_where, travel_when, travel_stay_at. Then, the above triple is represented as a bunch of triples:

John ***travel_who travelC***

travelC travel_where *Tokyo*

travelC travel_when *2015-01-15*

travelC travel_stay_at *Hayat*

As we can observe, the property **travel** has been replaced by the class **travelC**. This new property class is described using three items (that play the role of objects) in new triples with **travelC** as their object. Please note, that new properties have been introduced to describe the property.

4.2. Linguistically Oriented Interface for Querying RDF Data

A specialized interface is required to query RDF data using imprecise terms describing temporal aspects of data, as well as to use imprecise quantitative descriptors that impose additional constrains on the results of queries. Such an interface – called Linguistically Oriented Interface (LORI) – is presented here. It is built based on RDF/RDFS reasoner [15] provided by Jena [14] that works with a set of basic entailments rules, Section 4.2.1 Jena's RDF/RDFS reasoner allows for developing custom built-in predicates. A set of predicates that allow for querying temporal aspects of data, and processing complex data has been developed.

The above-mentioned predicates as well as entailment rules of RDF/RDFS reasoner are used to construct high-level predicates that provide the users with simpler and more straightforward ways of making inquiries. A set of mapping rules is used as a translator between the reasoner's predicates/rules and high-level predicates. A diagram representing architecture of LORI is shown in Fig 24. The main building blocks of LORI are: *CustomPredicates&RDFSchema* – a component containing developed predicates and required data structures; *ReasonerInterface* – an element providing an access to built-in custom predicates and RDF/RDFS reasoner's entailment rules; *UserInterface* – a set of high-level predicates available to the users; *MappingEngine* – a unit that uses mapping rules to translate queries built by the user to queries offered by *ReasonerInterface*.

4.2.1. Predicates, Data Structures and Reasoner Interface

A detailed description of LORI should start with an explanation and details regarding the *CustomPredicates&RDFSchema* component. However, before we can do this we should explain what type of input is acceptable for Jena's reasoner. Jena's RDF/RDFS reasoner works with entailment rules [15] in the form of **triple_patterns** [25]. These triples are executed against RDF data they identify RDF triples that match the 'fixed' positions (i.e., positions that contain explicit values) and have arbitrary values on the 'variable' positions (i.e., positions with a '?' character at their beginnings). For example, the **triple_pattern**:

```
(?subject  foaf:first_name  'John')
```

has the 'fixed' positions, foaf:first_name and 'John', and the variable position subject. As the result, we obtain a set of all entities (values of subject) that have the first name John, i.e., entities that are connected via the property foaf:first_name with 'John'. The **triple_pattern**s use entities defined within the considered data. Essentially, they are SPARQL graph (triple) patterns that are being matched against data graphs [25].

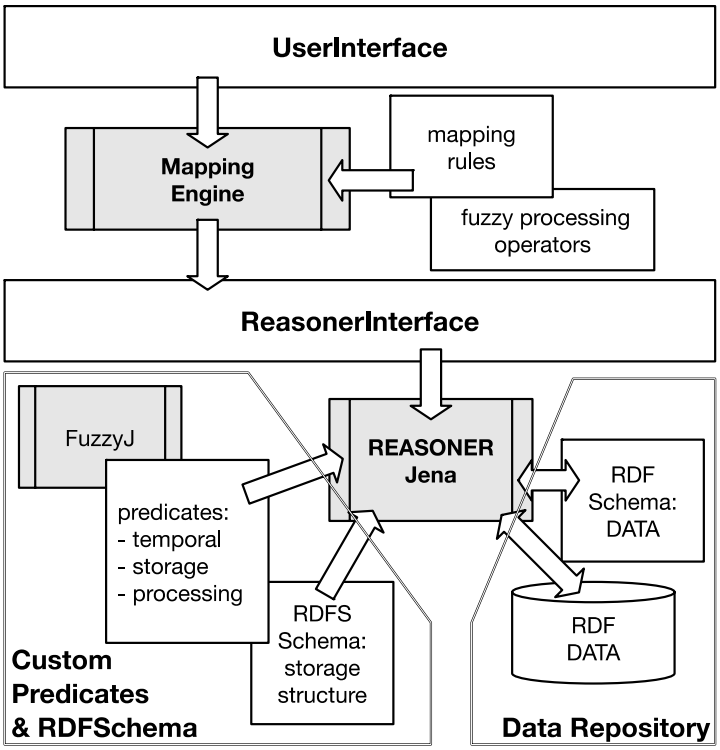


Figure 24 - Configuration of LORI

Processes of constructing queries in the case of temporal and complex data require knowledge and understanding of internal data structures. To address these needs we develop a number of predicates that simplify these processes. These predicates are built according to Jena's syntax and requirements imposed by RDF/RDFS reasoner. As we can see later, these predicates are further mapped into very simple and human-friendly high-level predicates that constitute *UserInterface*.

4.2.2. Fuzzy Temporal Predicates

A library of predicates capable of dealing with temporal instances and intervals is developed. Such a library gives the users the ability to build queries/rules that can express relations between temporal events and infer about them.

An important part required by these predicates is a concept of a time instant:

```
timeInstant = <specificTime|timeOfAnotherEvent>
```

The instant is given explicitly via the value of the parameter specificTime that can assume the values: NOW, YESTERDAY, LAST_SECOND, LAST_MINUTE, LAST_HOUR, LAST_DAY, or via providing date and time, such as "2012", "2012-05", "2012-05-06", or "2012-05-06 16:23".

The four defined and implemented predicates are:

approx_at_instant(?event[2], ?predicate, ?timeInstant, ?cut)

– to determine a degree of temporal overlapping between an event and an approximated instant in time – it calculates the level of degree to which the statement "something happened approximately around a given time instant" is true.

approx_in_interval(?event, ?predicate, interval_start, interval_end, ?cut)

– to determine a degree of overlapping between an event and an interval defined by starting and ending points.

approx_at_instant_before(?event, ?predicate, p, unit_p, ?timeInstant, ?cut)

– to determine a degree of overlapping between an event and an instant that occurs approximately a number of time granules before some instant of time. For example, the predicate approx_at_instant_before(?event, $property 2, days, ?time-Instant, ?degree) calculates the degree of overlapping of an event with an instant at about 2 days before timeInstant.

approx_in_interval_before(?event, ?predicate, n, unit_n, ?timeInstant, m, unit_m, ?cut)

– to determine a degree of overlapping of an event with an interval that spans across n granules of time, and this interval occurs m granules of time before timeInstant.

The parameters of the predicates are:

[2] In SWRL, the character "?" is used to denote a variable.

`event` – a subject of RDF triple with `property` as its RDF property; if it is not given – nodes are identified by the parameter `predicate`;

`predicate` – an RDF predicate that is the focus of a temporal analysis (this `predicate` should have time as its range);

`timeInstant` – a moment in time defined in multiple ways based on temporal ontology;

`interval_start, interval_end` – two `timeInstant` values representing the beginning and the end of a time interval;

`p` – number of time units;

`unit_p` – granularity of time units for `p`;

`n` – number of time units, "width" of an interval;

`unit_n` – granularity of time units for `n`;

`m` – number of time units 'between' the end of the interval and `timeInstant`;

`unit_m` – granularity of time units for `m`;

`cut` – a value from the interval `<0,1>` indicating a desired degree to which the event should 'overlaps' with `timeInstant`, by default `cut` is equal to zero.

The predicates – available at *ReasonerInterface* – utilize fuzzy temporal ontology, Section 3, and use the FuzzyJ reasoner [19] suitable to deal with fuzzy data. The predicates are implemented as functions called from within Jena, and they further call FuzzyJ procedures to perform fuzzy calculations and reasoning. The same predicates with a reduced number of parameters are available at *UserInterface*, Section 4.2.2.

4.2.3. Storage Predicates

The storage predicates increase functionality of LORI by allowing the users to store the results of queries, reuse them, merge them, and perform operations, including fuzzy ones, on them. The results are stored in a form of RDF data. This allows for a full

integration of the results with RDF data that the LORI is working with. We have created an RDF Schema that defines classes and properties to build such RDF triples, Fig 25.

Three predicates for dealing with results of queries and use RDF Schema presented in Fig. 25 are developed. They are:

StoreResult(?resName) takes a single parameter resName as a name of RDF graph representing the results; this name is used for accessing the results in the future;

MergeResults(?resName1, ?resName2, ?resName3) used for merging two RDF graphs containing the results of two different queries, where resName1 is one of them, and resName2 is another, the result of the merge is stored in an RDF graph named resName3; and

ShowResult(?resName) used for displaying an RDF graphs containing results of a single query, where resName is its identifier.

These predicates constitute a very important part of the LORI – they allow for creating sequences of queries where the results of one query can be combined with the results of another query.

```
<rdfs:Class rdf:ID="QueryResult" />
<rdfs:Class rdf:ID="QueryResultNode" />
<rdf:Property rdf:ID="hasNode">
      <rdfs:domain rdf:resource="#QueryResult"/>
      <rdfs:range rdf:resource="#QueryResultNode"/>
</rdf:Property>
<rdf:Property rdf:ID="hasFuzzyNumber">
     <rdfs:domain rdf:resource="#QueryResultNode"/>
     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#decimal"/>
</rdf:Property>
<rdf:Property rdf:ID="hasSubject">
     <rdfs:domain rdf:resource="#QueryResultNode"/>
     <rdfs:range                  rdf:resource="http://www.w3.org/2000/01/rdf-
schema#Resource"/>
</rdf:Property>
```

```
<rdf:Property rdf:ID="hasPredicate">
     <rdfs:domain rdf:resource="#QueryResultNode"/>
     <rdfs:range                  rdf:resource="http://www.w3.org/2000/01/rdf-
schema#Resource"/>
</rdf:Property>
<rdf:Property rdf:ID="hasObject">
     <rdfs:domain rdf:resource="#QueryResultNode"/>
     <rdfs:range                  rdf:resource="http://www.w3.org/2000/01/rdf-
schema#Resource"/>
</rdf:Property>
```

Figure 25 - The RDF Schema for storing results

4.2.4. Fuzzy Processing Predicates

The introduced data structures for storing results have created an opportunity to design and develop special functions operating on the obtained results. An important set of useful processing utilities includes fuzzy processing predicates. These predicates are implemented as functions inside Jena. They are further used via experts to define processing operators. The definitions are included in a mapping file.

An example of a fuzzy processing predicate is `Most`. It performs fuzzification of data based on a defined fuzzy membership function. Section 4.2.5 contains details how it is applied to define a fuzzy processing operator, and how it is used together with storage predicates at *UserInterface*. Section 5 illustrates its application.

4.2.5. UserInterface and MappingEngine

Presented above predicates are still closely related to data, i.e., in order to use them the user has to know details regarding data structures. To avoid this, an expert in the data under consideration builds a set of high-level predicates that ‘isolate’ the user from data details and its complexity.

The expert constructs high-level predicates and mapping rules. The mapping rules define these predicates and specify a way of 'transforming' them into the predicates and entailment rules offered at *ReasonerInterface*, Section 4.2.1. *MappingEngine* performs such a translation process. An example of a mapping rule is shown below:

highLevel_pred^{i}*(parameter_set)*
-> triple_pattern$_1$|…| ***triple_pattern***$_N$

The high-level predicates **highLevel_pred**^{i} should reflect inquiries frequently made by the users, and be of high importance with meaningful naming and intuitively recognized effects. Their parameters become inputs to individual **triple_pattern**$_k$().

In the case of temporal predicates, Section 4.2.2, *UserInterface* offers the same predicates but with a reduced number of parameters. *UserInterface* temporal predicates are:

approx_at_instant*(?timeInstant, ?cut)*
approx_in_interval*(?interval_start, interval_end, ?cut)*
approx_at_instant_before*(?p, unit_p, ?timeInstant, ?cut)*
approx_in_interval_before*(?n, unit_n, ?timeInstant, m, unit_m, ?cut)*

Similarity, the storage predicates, *UserInterface* offers the same predicates as the ones defined at the level of *ReasonerInterface*, Section 4.2.3. The predicates **MergeResult** and **ShowResult** are the same, while the third predicate has an additional parameter:

StoreResult*(?resName, ?fuzzy_processing_operator*$_r$*)*

This new parameter fuzzy_processing_operator$_r$ is an operator defined by an

expert – also in a mapping rules file – in the following way:

FProcOperName(fuzzy_processing_operator$_r$) ->
FuzzyProcessingParam(Most, Subject)

It means that, the operator `fuzzy_processing_operator`$_r$ contains the predicate `Most`, and data entities on which the predicate works. In the case above, these are `Subject`s of RDF triples representing the stored results.

Fuzzy processing operators defined on processing predicates are very depended on the data under consideration. For specifics see Section 5.

4.3. Case Studies

4.3.1. "Travel" Data

This case study shows how to apply LORI to query data containing temporal information and quite complex data structure. The considered RDF data contains triples describing traveling facts: destinations and dates, of a number of individuals. Additionally, the data also includes details regarding their illnesses, i.e., names of diseases, and times when they had it. An example of this data for two individuals X and Y is shown in Fig 26.

As we can see, Fig 26, the description of a single trip contains a blank node (*N_* or *P_*). It is needed in order to express two features: destination of the travel, and date when it took place. Information about a single disease also contains a blank node (*M_* and *Q_*) to represent: disease name and a date when a person became ill.

Such a structure of data shows an extra complication in data structure – something we would like to 'hide' from the user. At the same time, this gives us an opportunity to show how LORI works and simplifies queries.

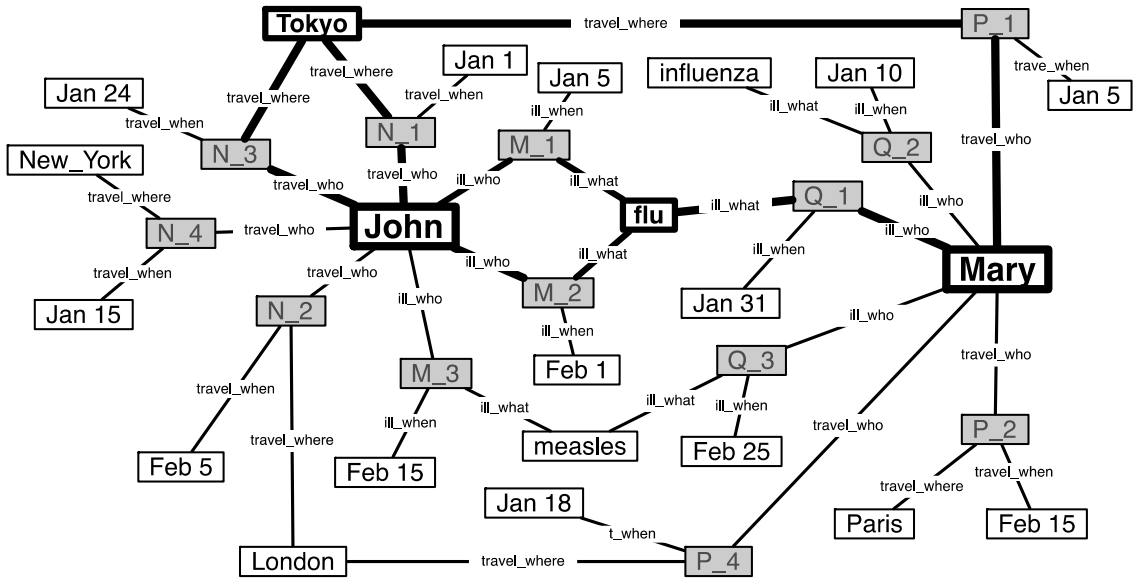


Figure 26 - RDF triples representing data two individuals X and Y (for simplify the properties do not have prefix rdfs:get_ used in the text below)

Complexity of data structure implies a need for high-level predicates. The mapping rules of three predicates that also work as their definitions are shown below:

```
WHO_HAS_DISEASE(.*1) =>
            (?Subject rdfs:get_ill_who ?interconnectNode),
            (?interconnectNode rdfs:get_ill_what ?*1)


WHO_TRAVEL_TO(.*1) =>
            (?Subject rdfs:get_travel_who ?interconnectNode),
            (?interconnectNode rdfs:get_travel_where ?*1)


WHERE_PERSON_TRAVELED(.*1) =>
            (?*1 rdfs:get_travel_who ?interconnectNode),
            (?interconnectNode rdfs:get_travel_where ?Object)
```

The predicate **WHO_HAS_DISEASE**(.*1), where *1 represents an input parameter that should be a person, is mapped into two RDF triple patterns. The first of them returns all Subjects – persons in our case – and interconnectNodes – blank nodes – that are

connected via the property rdfs:get_ill_who. The second one takes the identified interconnectNodes and looks for any entities that are connected to it via the property rdfs:get_ill_what. In this way, we obtain a list of all persons – values of Subjects - who had the disease identified by the input parameter *1.

The second predicate **WHO_TRAVEL_TO**(.*1) works in the similar way, but as the result we obtain a list of persons who travelled to the destination identified by *1.

For the predicate **WHERE_PERSON_TRAVELED**(.*1) the first pattern leads to pairs: <*1 interconnectNode> that are connected via rdfs:get_travel_who. This provides all blank nodes connected to a given person – identified by *1. Then we 'follow' the blank nodes and obtain Objects, i.e., places to which the person *1 travelled.

For queries about temporal aspects as well as for storing results, we use the temporal and storage predicates described in Section 4.2.1. For the case of fuzzy processing operators, we have mentioned that they are data dependent, so in this particular case the expert defines three operators that can be used as an input parameter for the predicate

StoreResult*(?resName, ?fuzzy_processing_operator*$_r$*)*

they are:

*FProcOperName(**MostWho**)=> FuzzyProcessingParam(Most, Subject)*

*FProcOperName(**MostWhere**)=>FuzzyProcessingParam(Most, Object)*

*FProcOperName(**Most**)=> FuzzyProcessingParam(Most, Both)*

In this case, fuzzy_processing_operator$_r$ can assume values MostWho, MostWhere, Most. Each of them works on different parts of RDF triples: the first one on subject, the second on object, and the third on both.

At this stage we can also show how the predicate **StoreResult**() with a fuzzy processing operator is translated into triples and rules of *ReasonerInterface*:

```
StoreResult(?resName, MostWho) =>
=> StoreResult(resName_temp)
=> (resName_temp rdf:type queryResult) -> Result(?resName, Most,
```

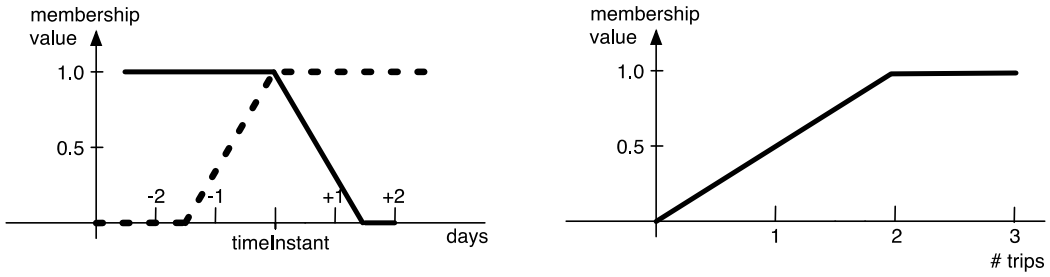
```
Subject)
```

As it can be seen, the predicate is translated into *ReasonerInterface* predicate StoreResult(), and the rule that allows RDF/RDFS reasoner to perform fuzzy processing.

In the first query we identify all individuals who traveled most often to Tokyo in mid-January 2015, who had flu one week after traveling to Tokyo, over the period of about two weeks. The queries look like this:

```
WHO_TRAVEL_TO('Tokyo'),
approx_in_interval('2015-01-01', '2015-31-01'),
-> StoreResult(ResultT, MostWho)
WHO_HAS_DISEASE('flu'),
approx_in_interval('2015-01-22', '2015-02-11'),
-> StoreResult(ResultD)
MergeResult(ResultT, ResultD, Result)
ShowResult(Result)
```

The *UserInterface* predicate **WHO_TRAVEL_TO**, together with a temporal predicate – approx_in_interval() is executed first. This results in a list of individuals – a single entry for each trip – who traveled to Tokyo. The travel takes place approximately between January 1st and 31st. The boundaries of the interval are 'modified' using fuzzy membership functions shown in Fig. 27 (a). The domain of this function is timeInstant. For the second predicate, the process is quite similar; it selects individuals who had flu over the period of three weeks – from January 22nd to February 11th, approximately.



(a) (b)

Figure 27 - Membership functions: (a) modifiers of interval's boundaries to make them approximate, and (b) for the predicate Most

The results of the first predicate are stored under the name ResultT. They are processed using the fuzzy processing operator MostWho to identify individuals who traveled to Tokyo most often. In other words, the results indicate how many times a person travelled to Tokyo, and this number is evaluated using the fuzzy membership function associated with the predicate Most, Fig. 27 (b). The results of the second high-level predicate are stored with the name ResultD. They represent all individuals who had flu approximately between Jan 22nd and Feb 11th, 2015.

The predicate **MergeResult**() combines both results. The fusion is based on the individuals – so, the persons who travelled the most to Tokyo and then had flu in the identified period of three weeks. A sample of the final results is shown below:

```
…
ResultT(1): John (Tokyo) -> temp: 1.0 trips: 1.0
ResultT(2): Mary (Tokyo) -> temp: 1.0 trips: 0.5
ResultD(1): John (Flu) -> temp: 1.0
ResultD(2): Mary (Flu) -> temp: 1.0
Result(1): John (Flu/Tokyo) -> temp: 1.0 trips: 1.0
Result(2): Mary (Flu/Tokyo) -> temp: 1.0 trips: 0.5
…
```

As we can see, John who traveled to Tokyo at least 2 times, Fig 27 (b), and had a flu has been identified as a person who match the query to the highest degree. The part temp: indicates degree of satisfaction of the temporal requirement, trips: - satisfaction of the fuzzy operator MostWho.

The second query should identify individuals who had measles at the beginning of May, and who traveled the most over the second week of May.

```
WHO_HAS_DISEASE('measles'),
approx_at_instant('2015-06-01'),
-> StoreResult(ResultD)
WHERE_PERSON_TRAVELED(),
approx_in_interval('2015-05-07', '2015-05-14', 0.75),
-> StoreResult(ResultP, MostWhere)
MergeResult(ResultD, ResultP, Result)
```

```
ShowResult(Result)
```

The results, shown below, indicate that the query is not successful. The first predicate provides a list of *persons* who had measles around May 1st, while the second gives a list of *places* visited by individuals – here Susan was in Toronto at least twice (trips: 1.0). Both queries have different types of entities as their responses and the result of merge has zero entries.

```
…
ResultD(1): John (Measles) -> (time) 1.0
ResultD(2): Mary (Measles) -> (time) 1.0
ResultD(3): Susan (Measles) -> (time) 1.0
ResultP(1): Toronto (Susan) -> temp: 1.0 trips: 1.0
ResultD(2): Calgary (Paul) -> temp: 1.0 trips: 0.5
…
```

4.3.2. DBLP Data

In order to illustrate the query mapping system and the proposed approach to query RDF data with temporal and fuzzy terms we use the DBLP database (http://dblp.uni-trier.de/db/). In particularly, we focus on a portion of the database – publication records for the years 1993-1999. RDF-XML files with the records have been downloaded. Based on the vocabulary used in the files we have created an RDF Schema (required by the RDF reasoner). An example of the schema with the most important elements is shown in Fig. 28.1.

```
<rdf:Property rdf:ID="has-date">
     <rdfs:domain rdf:resource="#Article-Reference"/>
     <rdfs:range rdf:resource="#Calendar-Date"/>
</rdf:Property>
<rdf:Property rdf:ID="has-web-address">
     <rdfs:domain rdf:resource="#Article-Reference"/>
     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</rdf:Property>
<rdf:Property rdf:ID="has-volume">
     <rdfs:domain rdf:resource="#Article-Reference"/>
     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#double"/>
</rdf:Property>
     <rdf:Property rdf:ID="article-of-journal">
     <rdfs:domain rdf:resource="#Article-Reference"/>
```

```
     <rdfs:range rdf:resource="#Journal"/>
</rdf:Property>
<rdf:Property rdf:ID="has-title">
     <rdfs:domain rdf:resource="#Article-Reference"/>
     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</rdf:Property>
<rdf:Property rdf:ID="has-author">
     <rdfs:domain rdf:resource="#Article-Reference"/>
     <rdfs:range rdf:resource="#Person"/>
</rdf:Property>
<rdf:Property rdf:ID="full-name">
     <rdfs:domain rdf:resource="#Person"/>
     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</rdf:Property>
```

Figure 28.1 - A fragment of the RDF Schema for DBPL

A very small example of the data is shown in Fig. 28.2. It represents an entry related to a single publication.

```
<rdf:Description rdf:about
       ="http://dblp.rkbexplorer.com/id/journals/jei/SinhaGLSM96">
     <rdfs:has-author rdf:resource
       ="http://dblp.rkbexplorer.com/id/people-dcc8bba393b2bd1c30db2c44765f0fa4-
                 64465f01287e3deb3f6c940f9d4f8e20"/>
     <rdf:type rdf:resource
       ="http://www.aktors.org/ontology/portal#Article-Reference"/>
     <rdfs:has-author rdf:resource
       ="http://dblp.rkbexplorer.com/id/people-7ee2fa9aff5bbfd9f5f5152865b9b892-
                4bb329520237c41ff83bc28230bfce22"/>
     <rdfs:has-date rdf:resource
       ="http://www.aktors.org/ontology/date#1996"/>
     <rdfs:article-of-journal rdf:resource
       ="http://dblp.rkbexplorer.com/id/journals-97b83d9eb09081a97c187d7289b2d5fc"/>
     <rdfs:has-author rdf:resource
       ="http://dblp.rkbexplorer.com/id/people-947c115c9d544b6616049a4415bf3e54-
                017c5bf8ceab4ee468822c45696a98ab"/>
     <rdfs:has-author rdf:resource
       ="http://dblp.rkbexplorer.com/id/people-74824503207e2826d0c8230cf74f30a7-
                16ce4b551357a941b11fd0f51581ece0"/>
     <rdfs:has-title>
           Classification and overview of research in real-time imaging.
     </rdfs:has-title>
     <rdfs:has-web-address>
           http://dx.doi.org/10.1117/12.245842
     </rdfs:has-web-address>
     <rdfs:has-volume>5</rdfs:has-volume>
     <owl:sameAs rdf:resource
     ="http://dblp.l3s.de/d2r/resource/publications/journals/jei/SinhaGLSM96"/>
     <rdfs:has-author rdf:resource
       ="http://dblp.rkbexplorer.com/id/people-483d0c940233f2cd118a64a930160151-
         5b9a283d0775e4fa85406b5f25ec5d34"/>
  </rdf:Description>
```

Figure 28.2 - RDF info about a single publication: Article-Reference

As it has been described above, the user's level query hides details of RDF data representation. This means that for each type of data there is a need to create high level predicates that are mapped into a series of low level predicates and entailment rules of the RDFS reasoner. Such a process has been done here. The two sample user-level queries are:

```
WHO_PUBLISHED(.*1)
 =>(?MainSubject rdfs:has-author ?interconnectNode),
   (?interconnectNode rdfs:full-name ?*1)

WHO_PUBLISHED_WITH(.*1)
 =>(?interconnectNode rdfs:has-author ?author1),
   (?author1 rdfs:full-name ?*1),
   (?interconnectNode rdfs:has-author ?author2),
   (?author2 rdfs:full-name ?MainSubject)
```

The first of them WHO_PUBLISHED(.*1) is mapped into two simple RDF level rules. One of them matches ids of papers with ids of people who are authors of these papers (property rdfs:has-author). The other one provides full name of these people (rdfs:full-name). If an input parameter *1 is provided – it is a full name of a person – then this simple RDF rule filters all papers – only papers with *1 as the author are obtained as the result.

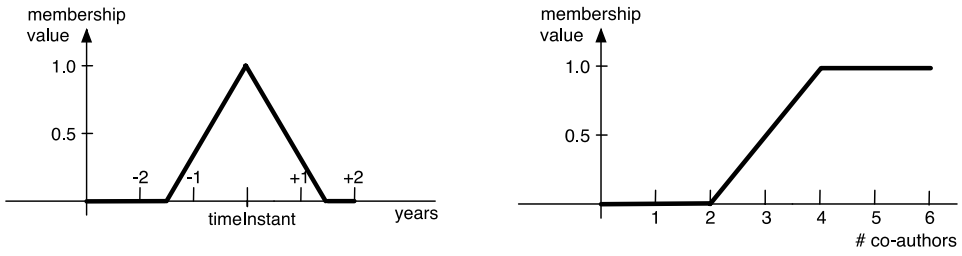
The second query is a bit more complicated. It is a combination of two "versions" of the previous query. The first part provides a list of papers and their first authors, or if the person's name *1 is given, it returns papers written by *1. The second part matches people to the papers identified in the first part of this query, and provide names of the papers' co-authors. As the result we obtain a list of peoples where each entry (for each person) contains all her publications and for each publication a list of its all co-authors.

The query performed on the RDF version of DBLP database is shown below:

```
WHO_PUBLISHED_WITH(),
approx_at_instant('1996','has-date'),
-> StoreResult(testResult, large, 'has-author')
```

```
ShowResult(testResult)
```

It uses the user's level predicate WHO_PUBLISHED_WITH() (Section VII.B), together with a temporal predicate – approx_at_instant() (Section V.B). The first part WHO_PUBLISHED_WITH() results in the list of authors and their publications together with co-authors. The second part approx_at_instant('1996', 'has-date', ?satDegree) "filters" the list based on the temporal requirement. Here, we specify the year 1996 so the predicate approx_at_instant() selects only triples with a value of the RDF predicate 'has-date' equals approximately 1996, i.e., using a fuzzy term approx 1996. This term can be defined arbitrary; in our implementation it has been defined in a way that the provided value of the parameter timeInstant (Section V.B) is used to build a fuzzy membership function, Fig. 28.3. The domain of this function is timeInstant +/- 1.5 years.



(a) (b)

Figure 28.3 - Membership functions of terms approx (a), and large (b)

The results are stored under the name testResult. The results are filtered with the term large on the predicate 'has-author'. This means that the number of co-authors on a given paper is "evaluated" using the fuzzy membership function associated with the label large, Fig 28.3.

The very last predicate ShowResult() displays a list of authors, coauthors and previously stored under the name testResult. A sample of the results is shown below:

```
…
testResult: "Sergey Gorinsky" -> 1.0 (5.0)
testResult: "D. Prangenberg" -> 0.5 (3.0)
testResult: "Alexander A. Zelensky" -> 1.0 (6.0)
testResult: "J. Andrew Bangham" -> 1.0 (4.0)
testResult: "Ursula Bernhard" -> 0.0 (1.0)
testResult: "Jacques G. Trecat" -> 0.0 (2.0)
testResult: "Yoichi Miyake" -> 1.0 (4.0)
…
```

Every entry of the results contains its id: testResult, name of the author in "", a degree to which this author satisfies the statement “an author of a paper with large number of co-authors, and the actual number of co-authors.

5. Implementation Details and Performance Evaluation

5.1. Used Software Packages and Libraries

5.1.1. Jena

For implementing LORI we used a software package Jena [14] that is an open source Semantic Web framework. Jena provides an API for reading and writing RDF graphs in different serialization formats such as RDF/XML, Turtle, and Notation 3. The class Model is the main class for representing an RDF graph. It is meant to have an interface with methods to help writing RDF-base applications. In general there are three RDF container concepts in Jena:

- graph: a mathematical view of the directed relations between nodes in a connected structure
- Model: a rich Java API with many convenience methods for Java application developers
- Graph: a simpler Java API intended for extending Jena's functionality.

The Jena packages that have been used in LORI implementation are listed in Table 2.

Table 2 - Jena packages used in LORI implementation

Package	Description
chh.jena.rdf.model	The Jena core. Creating and manipulating RDF graphs.
oaj.riot	Reading and Writing RDF.
chh.jena.datatypes	Provides the core interfaces through which datatypes are described to Jena.
chh.jena.ontology	Abstractions and convenience classes for accessing and manipulating ontologies represented in RDF.
chh.jena.rdf.listeners	Listening for changes to the statements in a model
chh.jena.reasoner	The reasoner subsystem is supports a range of inference engines which derive additional information from an RDF model
chh.jena.shared	Common utility classes
chh.jena.vocabulary	A package containing constant classes with predefined constant objects for classes and properties defined in well-known vocabularies.

Jena has an inference subsystem that allows for 'plugging-in' multiple different inference engines or reasoners. The subsystem is used for entailing new RDF assertions based on rules, exiting RDF data, and descriptions of ontology/RDF classes. The Jena's Inference System has a general rule engine that can be used for entailing any RDF data. The overall architecture of Jena inference subsystem is presented in Fig. 29.

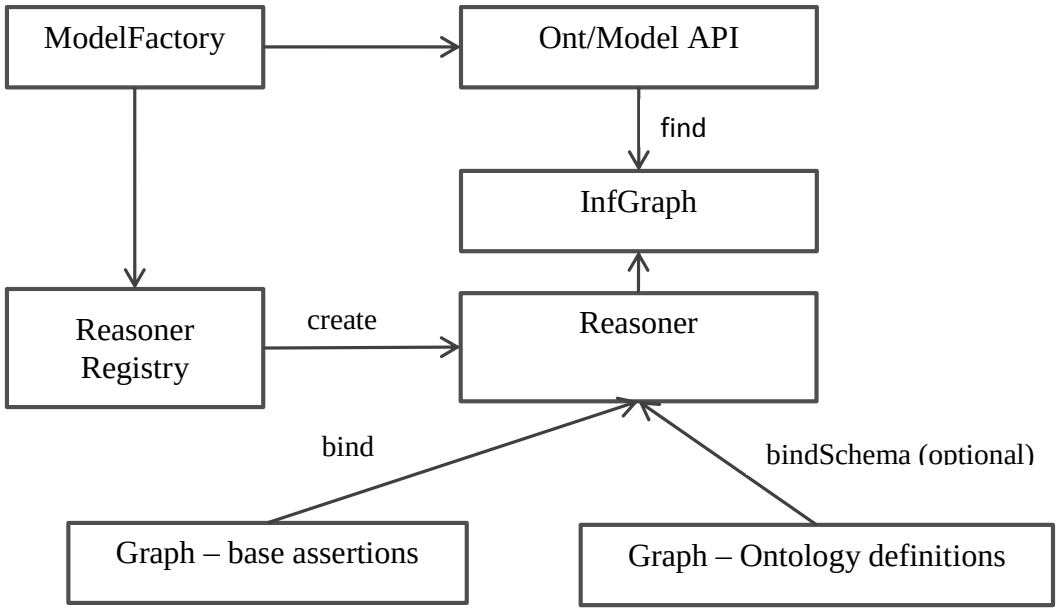


Figure 29 - Jena Inference Subsystem

ModelFactory is used to link data sets with reasoners at the time a new Model is created. *ModelFactory* allows applications to access Jena inference mechanism. The mechanism returns not only original data statements but also statements that are derived from data based on provided rules. Another component – *ReasonerRegistery* – is used for registering new reasoners types. In this way, Jena makes the Inference Subsystem open to interaction with reasoners other than predefined reasoners. Predefined reasoners in Jena are:

- Transitive reasoner: provides support for storing and traversing class and property lattices; it implements just the *transitive* and *reflexive* properties of rdfs:subPropertyOf and rdfs:subClassOf.
- RDFS rule reasoner: implements a configurable subset of the RDFS entailments.
- OWL, OWL Mini, OWL Micro Reasoners: a set of useful but incomplete implementation of a subset of the OWL/Full language.
- Generic rule reasoner: a rule based reasoner that supports user defined rules; additionally it supports forward chaining, tabled backward chaining and hybrid execution strategies, [15].

Jena RDFS reasoner supports almost all RDFS entailments, Table 3b. It can be configured to work at three different levels (modes): Full, Default and Simple. In Full mode RDFS reasoner covers all RDFS axioms and closure rules except bNode entailment. The Default Mode removes "everything is a resource" and "everything used as a property is one" (rdf1, rdfs4a and rdfs4b in Table 3a and 3b) and expensive checks for container membership properties. The Simple mode covers just the transitive closure of subPropertyOf and subClassOf relations, the domain and range entailments and the implications of subPropertyOf and subClassOf.

The RDFS entailment holds for all the following patterns, which correspond closely to the RDFS semantic conditions, Table 3.

Table 3a - RDF entailments

Rule Name	if E contains	then add
rdf1	uuu aaa yyy .	aaa rdf:type rdf:Property .
rdf2	uuu aaa lll . where lll is a well-typed XML literal .	_:nnn rdf:type rdf:XMLLiteral . where _:nnn identifies a blank node allocated to lll by rule lg

Table 3b - RDFS entailments

Rule Name	If E contains:	then add:
rdfs1	uuu aaa lll. where lll is a plain literal (with or without a language tag).	_:nnn rdf:type rdfs:Literal . where _:nnn identifies a blank node allocated to lll by rule rule lg.
rdfs2	aaa rdfs:domain xxx . uuu aaa yyy .	uuu rdf:type xxx .
rdfs3	aaa rdfs:range xxx . uuu aaa vvv .	vvv rdf:type xxx .
rdfs4a	uuu aaa xxx .	uuu rdf:type rdfs:Resource .
rdfs4b	uuu aaa vvv.	vvv rdf:type rdfs:Resource .
rdfs5	uuu rdfs:subPropertyOf vvv . vvv rdfs:subPropertyOf xxx .	uuu rdfs:subPropertyOf xxx .
rdfs6	uuu rdf:type rdf:Property .	uuu rdfs:subPropertyOf uuu .
rdfs7	aaa rdfs:subPropertyOf bbb . uuu aaa yyy .	uuu bbb yyy .
rdfs8	uuu rdf:type rdfs:Class .	uuu rdfs:subClassOf rdfs:Resource .
rdfs9	uuu rdfs:subClassOf xxx . vvv rdf:type uuu .	vvv rdf:type xxx .
rdfs10	uuu rdf:type rdfs:Class .	uuu rdfs:subClassOf uuu .
rdfs11	uuu rdfs:subClassOf vvv . vvv rdfs:subClassOf xxx .	uuu rdfs:subClassOf xxx .
rdfs12	uuu rdf:type rdfs:ContainerMembershipProperty .	uuu rdfs:subPropertyOf rdfs:member .
rdfs13	uuu rdf:type rdfs:Datatype .	uuu rdfs:subClassOf rdfs:Literal .

The whole process of using Jena can be summarized in following steps: first step is creating RDF model which can be done by using *ModelFactory.createRDFSModel*. Next step is configuring the reasoner. A Jena resource object can be passed to

ModelFactory.Create method for configuring behavior of Jena reasoners. The created reasoner will be configured by the properties of passed object. Another way to configure Jena reasoner is using *Reasoner.SetParameter* that is used for setting just a single configuration parameter. The next step is creating an inference model by attaching the reasoner to a set of RDF data. It depends on the implementation strategy to put all data in one model or separate them into schema and instance data. If you want to use a schema definition for more than one instance data, so it is better to create separate models. *Reasoner.bindSchema* is used for binding a RDF schema to reasoner and *ModelFactory.createInfModel* is used for binding data to reasoner and crating inference model.

5.1.2. Jena Built-Ins

Jena built-ins are procedural Java pieces of code that can be used in Jena rules as predicates performing various tasks. They can appear either in the rules' body, head or both. There are some predefined primitive built-ins that are stored in a reasoner's registry (Fig. 29). Each built-in returns either a true or false value as the result. Therefore, if a built-in is used in the body of a rule, and the result of this built-in is false then the head of rule will not be executed (the rule will not fire). A list of primitive built-ins is shown in Table 4.

Table 4 - Jena – a list of Primitive Built-in Functions

Builtin	Operations
isLiteral(?x) notLiteral(?x) isFunctor(?x) notFunctor(?x) isBNode(?x) notBNode(?x)	Test whether the single argument is or is not a literal, a functor-valued literal or a blank-node, respectively.
bound(?x...) unbound(?x..)	Test if all of the arguments are bound (not bound) variables
equal(?x,?y) notEqual(?x,?y)	Test if x=y (or x != y). The equality test is semantic equality so that, for example, the xsd:int 1 and the xsd:decimal 1 would test equal.
lessThan(?x, ?y),	Test if x is <, >, <= or >= y. Only passes if both x and y are

greaterThan(?x, ?y) **le(?x, ?y), ge(?x, ?y)**	numbers or time instants (can be integer or floating point or XSDDateTime).
sum(?a, ?b, ?c) **addOne(?a, ?c)** **difference(?a, ?b, ?c)** **min(?a, ?b, ?c)** **max(?a, ?b, ?c)** **product(?a, ?b, ?c)** **quotient(?a, ?b, ?c)**	Sets c to be (a+b), (a+1) (a-b), min(a,b), max(a,b), (a*b), (a/b). *Note that these do not run backwards, if in sum a and c are bound and b is unbound then the test will fail rather than bind b to (c-a). This could be fixed.*
strConcat(?a1, .. ?an, ?t) **uriConcat(?a1, .. ?an, ?t)**	Concatenates the lexical form of all the arguments except the last, then binds the last argument to a plain literal (strConcat) or a URI node (uriConcat) with that lexical form. In both cases if an argument node is a URI node the URI will be used as the lexical form.
regex(?t, ?p) **regex(?t, ?p, ?m1, .. ?mn)**	Matches the lexical form of a literal (?t) against a regular expression pattern given by another literal (?p). If the match succeeds, and if there are any additional arguments then it will bind the first n capture groups to the arguments ?m1 to ?mn. The regular expression pattern syntax is that provided by java.util.regex. Note that the capture groups are numbered from 1 and the first capture group will be bound to ?m1, we ignore the implicit capture group 0 which corresponds to the entire matched string. So for example regexp('foo bar', '(.) (.)', *?m1, ?m2)* *will bind m1 to "foo" and m2 to "bar".*
now(?x)	Binds ?x to an xsd:dateTime value corresponding to the current time.
makeTemp(?x)	Binds ?x to a newly created blank node.
makeInstance(?x, ?p, ?v) **makeInstance(?x, ?p, ?t, ?v)**	Binds ?v to be a blank node which is asserted as the value of the ?p property on resource ?x and optionally has type ?t. Multiple calls with the same arguments will return the same blank node each time - thus allowing this call to be used in backward rules.
makeSkolem(?x, ?v1, ... ?vn)	Binds ?x to be a blank node. The blank node is generated based on the values of the remain ?vi arguments, so the same combination of arguments will generate the same bNode.
noValue(?x, ?p) **noValue(?x ?p ?v)**	True if there is no known triple (x, p,) or (x, p, v) in the model or the explicit forward deductions so far.
remove(n, ...) **drop(n, ...)**	Remove the statement (triple) which caused the n'th body term of this (forward-only) rule to match. Remove will propagate

	the change to other consequent rules including the firing rule (which must thus be guarded by some other clauses). Drop will silently remove the triple(s) from the graph but not fire any rules as a consequence. These are clearly non-monotonic operations and, in particular, the behavior of a rule set in which different rules both drop and create the same triple(s) is undefined.
isDType(?l, ?t) notDType(?l, ?t)	Tests if literal ?l is (or is not) an instance of the datatype defined by resource ?t.
print(?x, ...)	Print (to standard out) a representation of each argument. This is useful for debugging rather than serious IO work.
listContains(?l, ?x) listNotContains(?l, ?x)	Passes if ?l is a list which contains (does not contain) the element ?x, both arguments must be ground, cannot be used as a generator.
listEntry(?list, ?index, ?val)	Binds ?val to the ?index'th entry in the RDF list ?list. If there is no such entry the variable will be unbound and the call will fail. Only useable in rule bodies.
listLength(?l, ?len)	Binds ?len to the length of the list ?l.
listEqual(?la, ?lb) listNotEqual(?la, ?lb)	listEqual tests if the two arguments are both lists and contain the same elements. The equality test is semantic equality on literals (sameValueAs) but will not take into account owl:sameAs aliases. listNotEqual is the negation of this (passes if listEqual fails).
listMapAsObject(?s, ?p ?l) listMapAsSubject(?l, ?p, ?o)	These can only be used as actions in the head of a rule. They deduce a set of triples derived from the list argument ?l : listMapAsObject asserts triples (?s ?p ?x) for each ?x in the list ?l, listMapAsSubject asserts triples (?x ?p ?o).
table(?p) tableAll()	Declare that all goals involving property ?p (or all goals) should be tabled by the backward engine.
hide(p)	Declares that statements involving the predicate p should be hidden. Queries to the model will not report such statements. This is useful to enable non-monotonic forward rules to define flag predicates that are only used for inference control and do not "pollute" the inference results.

A very important and powerful feature of Jena is its mechanism enabling development of custom built-in functions. These custom built-ins can be used in Jena rules in the same way like primitive built-ins. A custom built-in is an extension of the following Java class *com.hp.hpl.jena.reasoner.rulsys.builtins.BaseBuiltin*. It can return an

output value different than true/false. For this, an output parameter should be set inside a body of a built-in function. A sample code for creating a custom built-in is shown below. It is an implementation of function *POW(base, exponent)* that takes two parameters and returns raising *base* to the power *exponent*. This value is returned as an output parameter. As it is shown below, the result is bind to the second parameter.

```
package com.ge.research.sadl.jena.reasoner.builtin;

import com.hp.hpl.jena.graph.Node;
import com.hp.hpl.jena.reasoner.rulesys.BindingEnvironment;
import com.hp.hpl.jena.reasoner.rulesys.RuleContext;
import com.hp.hpl.jena.reasoner.rulesys.Util;
import com.hp.hpl.jena.reasoner.rulesys.builtins.BaseBuiltin;

public class Pow extends BaseBuiltin {
    public String getName() {
        return "pow";
    }

    public int getArgLength() {
        return 3;
    }

    public boolean bodyCall(Node[] args, int length, RuleContext context) {
        checkArgs(length, context);
        BindingEnvironment env = context.getEnv();
        Node n1 = getArg(0, args, context);
        Node n2 = getArg(1, args, context);
        if (n1.isLiteral() && n2.isLiteral()) {
            Object v1 = n1.getLiteralValue();
            Object v2 = n2.getLiteralValue();
            Node pow = null;
            if (v1 instanceof Number && v2 instanceof Number) {
                Number nv1 = (Number)v1;
                Number nv2 = (Number)v2;
                if (v1 instanceof Float || v1 instanceof Double
                ||  v2 instanceof Float || v2 instanceof Double) {
                  double pwd = Math.pow(nv1.doubleValue(),
nv2.doubleValue());
                    pow = Util.makeDoubleNode(pwd);
                } else {
                  long pwd = (long)
Math.pow(nv1.longValue(),nv2.longValue());
                    pow = Util.makeLongNode(pwd);
                }
                return env.bind(args[2], pow);
            }
        }
        return false;
    }
}
```

Figure 30 - Jena built-in: example of returning a value via parameter

All predicates – manifestations of built-ins at the level of rules – we have created in this project are custom built-in. In most cases, our built-ins return true as the result, while an actual result of a predicate is returned via an output parameter. For example, *approx_at_instant()* predicate takes at most four parameters, and one of them is actually an output variable that is set inside the body of the built-in. The built-in's code is:

```
package mine.BuiltIn.FuzzyTemporal;
import com.hp.hpl.jena.graph.Node;
import com.hp.hpl.jena.reasoner.rulesys.BindingEnvironment;
import com.hp.hpl.jena.reasoner.rulesys.BuiltinException;
import com.hp.hpl.jena.reasoner.rulesys.RuleContext;
import com.hp.hpl.jena.reasoner.rulesys.Util;
import com.hp.hpl.jena.reasoner.rulesys.builtins.BaseBuiltin;
import mine.JenaInferenceUse.RDF.Instant;
import mine.JenaInferenceUse.RDF.Interval;

public class approx_at_instant extends BaseBuiltin {
	@Override
	/**
	* Return a name for this built-in, normally this will be the name of
	the functor that will be used to invoke it.
	*/
	public String getName() {
		return "approx_at_instant";
	}

	/**
	* Return the expected number of arguments for this functor or 0 if the
	number is flexible.
	*/
	public int getArgLength() {
		return 4;
	}

	/**
	* @param args the array of argument values for the builtin, this is an
	array of Nodes, some of which may be Node_RuleVariables.
	* @param length the length of the argument list, may be less than the
	length of the args array for some rule engines
	* @param context an execution context giving access to other relevant
	data
	*/
	public boolean bodyCall(Node[] args, int length, RuleContext context)
	throws BuiltinException
	{
		TemporalHelper th = new TemporalHelper(context);

		boolean result = false;
		boolean hasUnbound1stArgument = false;
		double operationResult = 0.0;

		int numberOfArguments = args.length;
```

```
checkNumberOfArgumentsInRange(3, 4, numberOfArguments);

BindingEnvironment env = context.getEnv();
Node n0 = getArg(2, args, context);
if (n0.isVariable())
{
      hasUnbound1stArgument = true;
}

double argument0 = 0.0;
if (!hasUnbound1stArgument)
      try {
            argument0 = th
            .getArgumentAsADouble(2, getArg(2, args, context));
      } catch (FuzzyTemporalException e) {
            e.printStackTrace();
      }

Instant tempArgument3;
Interval argument2, argument3;
String argumentTimePredicate = "";

try {
      if(length == 4)
      {
            argumentTimePredicate = th.getArgumentAsAnString(3,
            getArg(3, args, context));
      }

      argument2 = th
            .getArgumentAsAnInterval(0, getArg(0, args, context),
            argumentTimePredicate, context);
      tempArgument3 = th
            .getArgumentAsAnInstant(1, getArg(1, args, context),
            context);

      argument3=new Interval(tempArgument3, tempArgument3, 2, 4);

      FuzzyJImp _fJ = new FuzzyJImp(argument2, argument3);
      operationResult = _fJ.getMaxOfIntersection();

} catch (FuzzyTemporalException e) {
      e.printStackTrace();
      // throw new BuiltinException(this, context, "Error : " +
      // e.getMessage());
      // FuzzyTemporalException("Error : " + e.getMessage());
}

if (hasUnbound1stArgument) {
      // Bind the result to the first argument.
      Node outputResult = null;
      outputResult = Util.makeDoubleNode(operationResult);
      env.bind(args[2], outputResult);

      result = true;
} else
      result = (argument0 == operationResult);
```

```
            return result;
        }

}
```

Figure 31 - Jena built-in: approx._at_instant()

Each built-in function should contain two methods: *getName* and *bodyCall*. The first method must return a name of this built-in function. That name is used as a predicate name in Jena rules. The second method contains implementation of the built-in. This implementation is invoked when a Jena rule containing this predicate is fired.

5.1.3. FuzzyJ

The package FuzzyJ is a set of Java classes used for handling fuzzy concepts and reasoning. Originally it is an extension of FuzzyCLIPS and it has been developed in National Research Council of Canada. The main classes of FuzzyJ are: FuzzyVariable, FuzzySet, FuzzyValue.

The FuzzyVariable class is used for creating a fuzzy universe of discourse, such as temperature or pressure. For defining a fuzzy variable its name (like temperature), units (like degrees C) and a universe of discourse (like a range of 0 to 100) are needed. Name and unit are just used for displaying, while a universe of discourse is defined as a set of lower and upper bounds for fuzzy sets defining a fuzzy variable. Following is an example of creating a fuzzy variable for temperature with universe of discourse from 0 to 100, and with a unit “C”:

```
FuzzyVariable temp
= new FuzzyVariable("temperature", 0, 100, "C");
```

The FuzzySet is a mapping of a set of real numbers to membership values in the range [0, 1]. It is represented by a set of pairs of u_i/x_i where x_i is the real number and u_i is a membership value associated with x_i. An example is {0.0/0.3, 1.0/0.5, 0.0/0.7} which represents a triangular fuzzy set. The following is code for creating this fuzzy set:

```
FuzzySet fSet = new triangleFuzzySet( 0.3, 0.5, 0.7 );
```

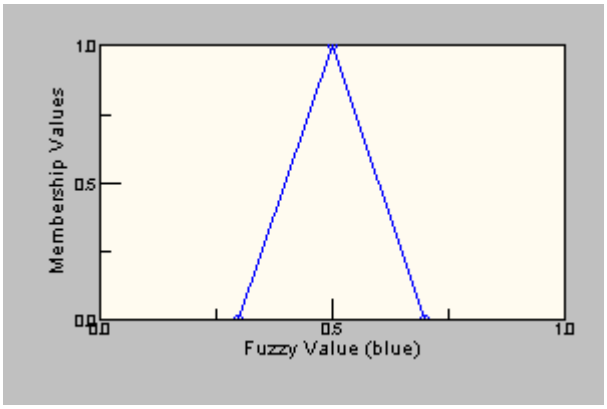


Figure 32 - Triangular fuzzy set

For creating a more complex fuzzy set we can use a FuzzySet constructor in the following way:

```
double yValues[] = {0, 1, 0.65, 1, 0};
double xValues[] = {0.1, 0.3, 0.4, 0.5, 0.8};
FuzzySet fSet = new FuzzySet( xValues, yValues, 5 );
```

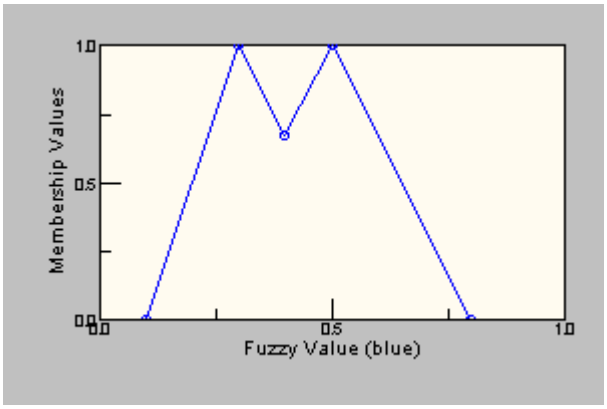


Figure 33 - 'Complicated' fuzzy set

FuzzyJ provides a set of subclasses for creating a fairly complete set of common fuzzy sets of different shapes. A hierarchy of fuzzy sets is shown in Fig. 34.

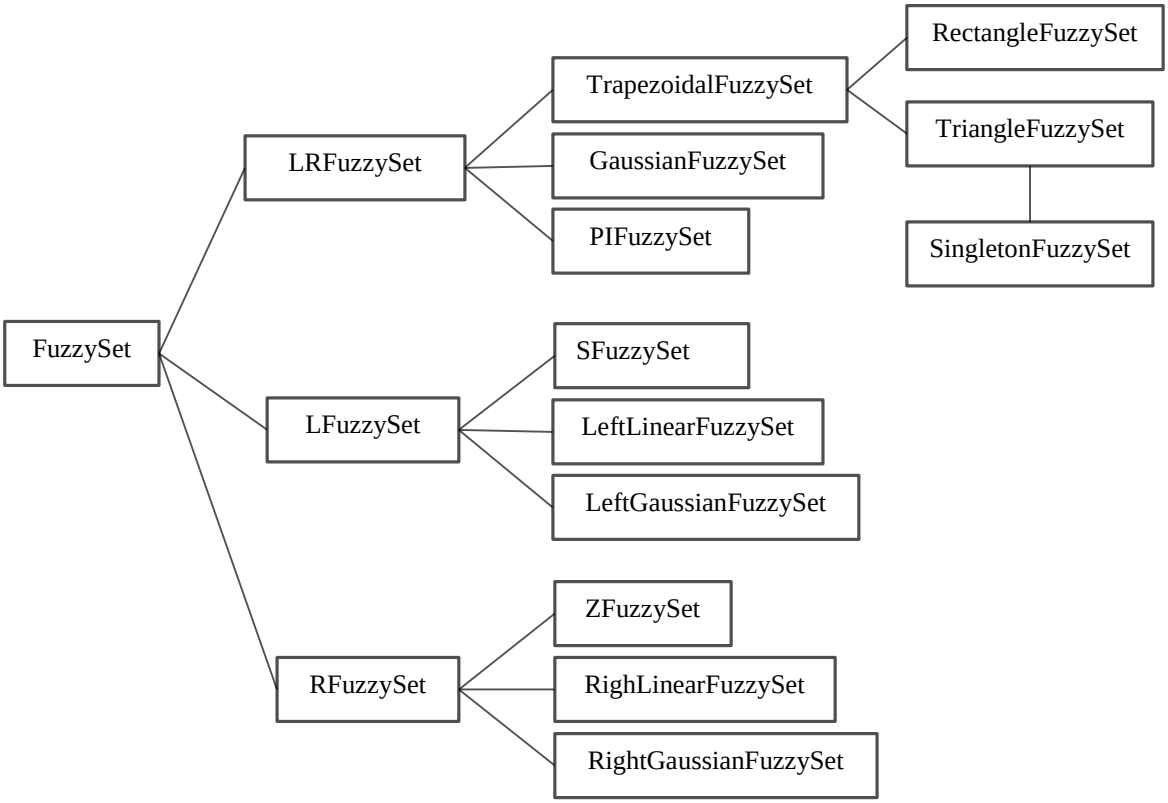


Figure 34 - Hierarchy of FuzzyJ's Fuzzy Sets

For example, the TriangleFuzzySet is sub-class of the TrapezoidFuzzySet. It is basically a trapezoid shape with its left and right sides connected at the same point. Also the TrapezoidFuzzySet is sub-class of the LRFuzzySet that has three different parts: left, right, and middle part. The left and right parts can have any various shapes and the middle part has always value set to 1. If the LRFuzzySet has linear shape on both left and right sides then it becomes to a TrapezoidFuzzySet.

FuzzyJ supports a range of operations on fuzzy sets. The most common are (for more FuzzyJ operations see [19]):

Fuzzy complement gives complement of fuzzy set. Mathematically it is:

$$(\text{NOT}),\ u_{compl}(x) = 1 - u(x),\ \text{or}\ y_{compl} = 1 - y.$$

Fuzzy intersection gives the intersection of two fuzzy sets. An intersection operator is similar to the logical operator AND it means that the membership (y) value at any x is the minimum of the membership values of the two fuzzy sets, Fig. 35.

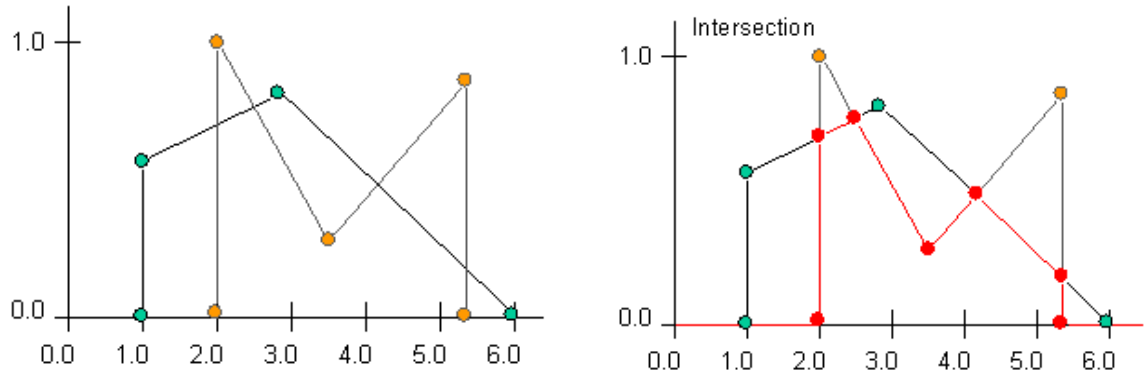


Figure 35 - Intersection of two fuzzy sets

Fuzzy union gives “a sum” of two fuzzy sets. Union operator is similar to logical operator OR where the membership (y) value at any x is the maximum of two membership values of the two fuzzy sets, Fig 36.

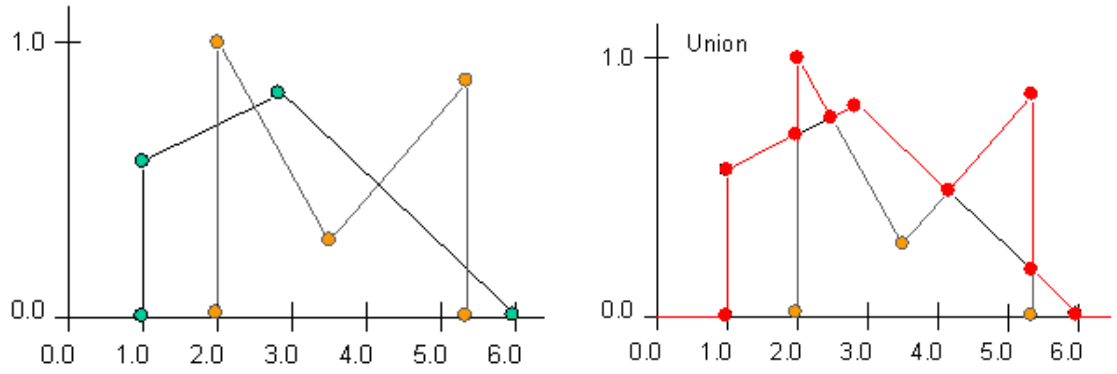


Figure 36 - Union of two fuzzy sets

Maximum of intersection gives the maximum membership value of the intersection set formed by tow fuzzy set, Fig. 37.

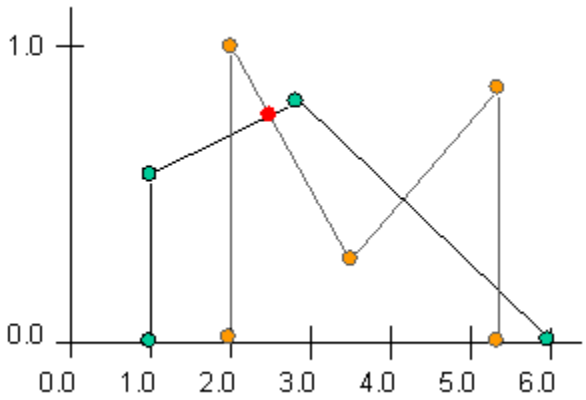


Figure 37 - Maximum of intersection of two fuzzy sets

5.2. Testing and Performance Validation of Developed Software

We used a real-world data obtained from Alberta Agriculture Department for testing our framework built using the proposed approach. The data set contains two years of data representing information about animal movements and diseases. The objective of performed data interaction and processing has been to track an animal disease as it moves between different counties, and find the county that was the origin of that disease.

5.2.1. Data Structures

The considered RDF dataset contains triples describing animal movements between counties in Alberta, the province of Canada. The data set has two main parts. The first one is about movement of animals from counties to auction centers. That data includes movement dates, kinds and numbers of moved animals. The second part is similar to the first part but it contains movement from auction places to target counties. The RDF dataset also contains information about animal diseases occurring in each county. That

data includes: dates when veterinarians visited farms, names of counties where farms are located, observed syndromes, clinical diagnosis, and number of affected animals.

The RDF schema is a meta-data that defines a vocabulary of concepts and terms used in the RDF data. It is shown in Fig. 38. It includes definitions of: classes of data, properties (relations) existing between classes and their types, domains and ranges, ways, and how classes and properties are related to each other.

RDF triples of each part of the dataset follow the pattern of defining N-ary relations, Section 4.1.2. It means that they have nodes representing relations. This allows for expressing some additional information describing these relations. In Fig. 38, we can observe such "relation nodes" as: "VPS_", "Manifest_" and "Permit_". "VPS_" is a node used to describe more information about an animal disease: county's name, date of occurrence, symptoms, disease name and affected animals. The "Manifest_" is used to better describe a movement of an animal from counties to auctions, while "Permit_" provides means to express information about animal movement from auction to counties dataset.

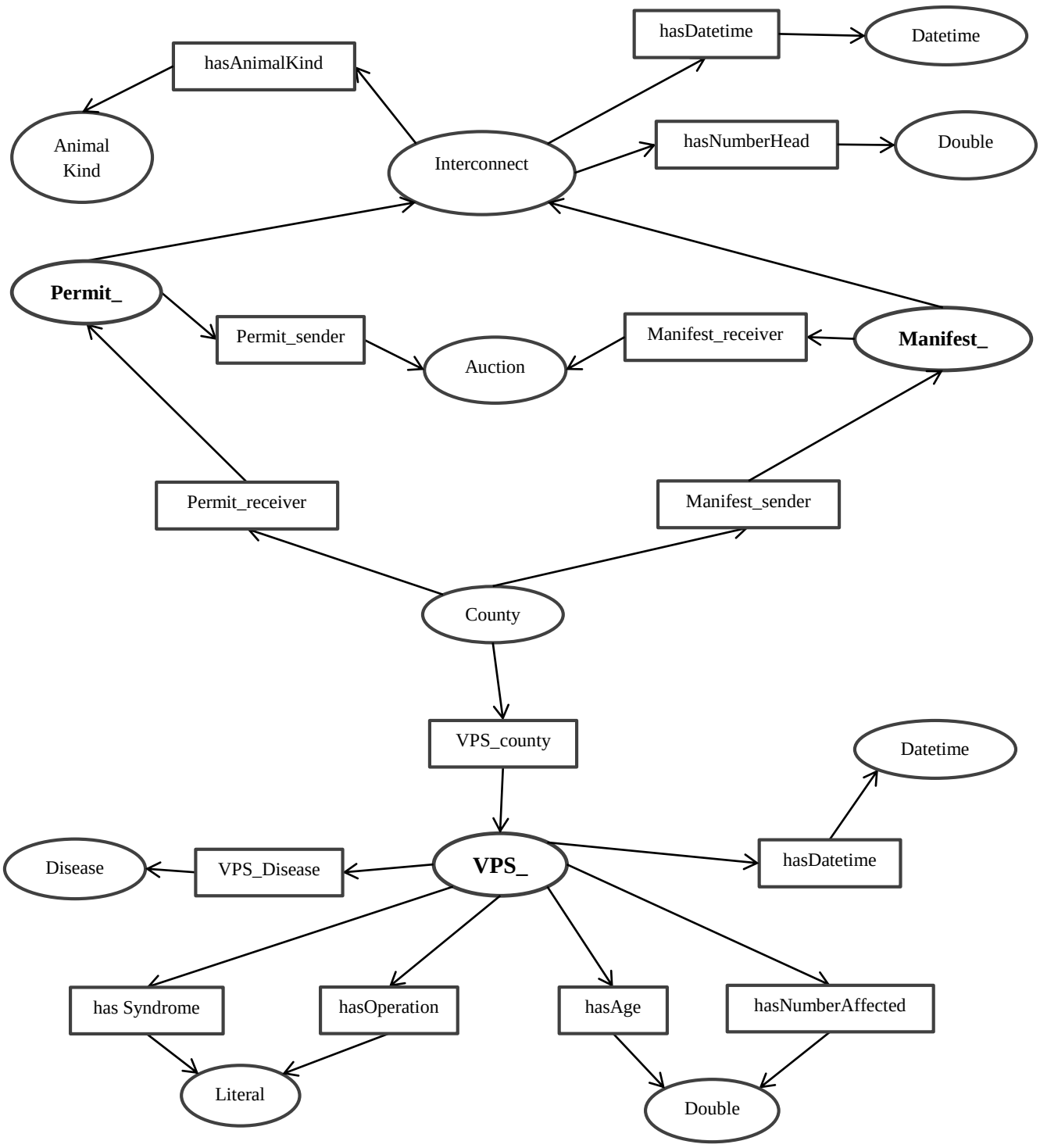


Figure 38 - RDFS for agriculture dataset

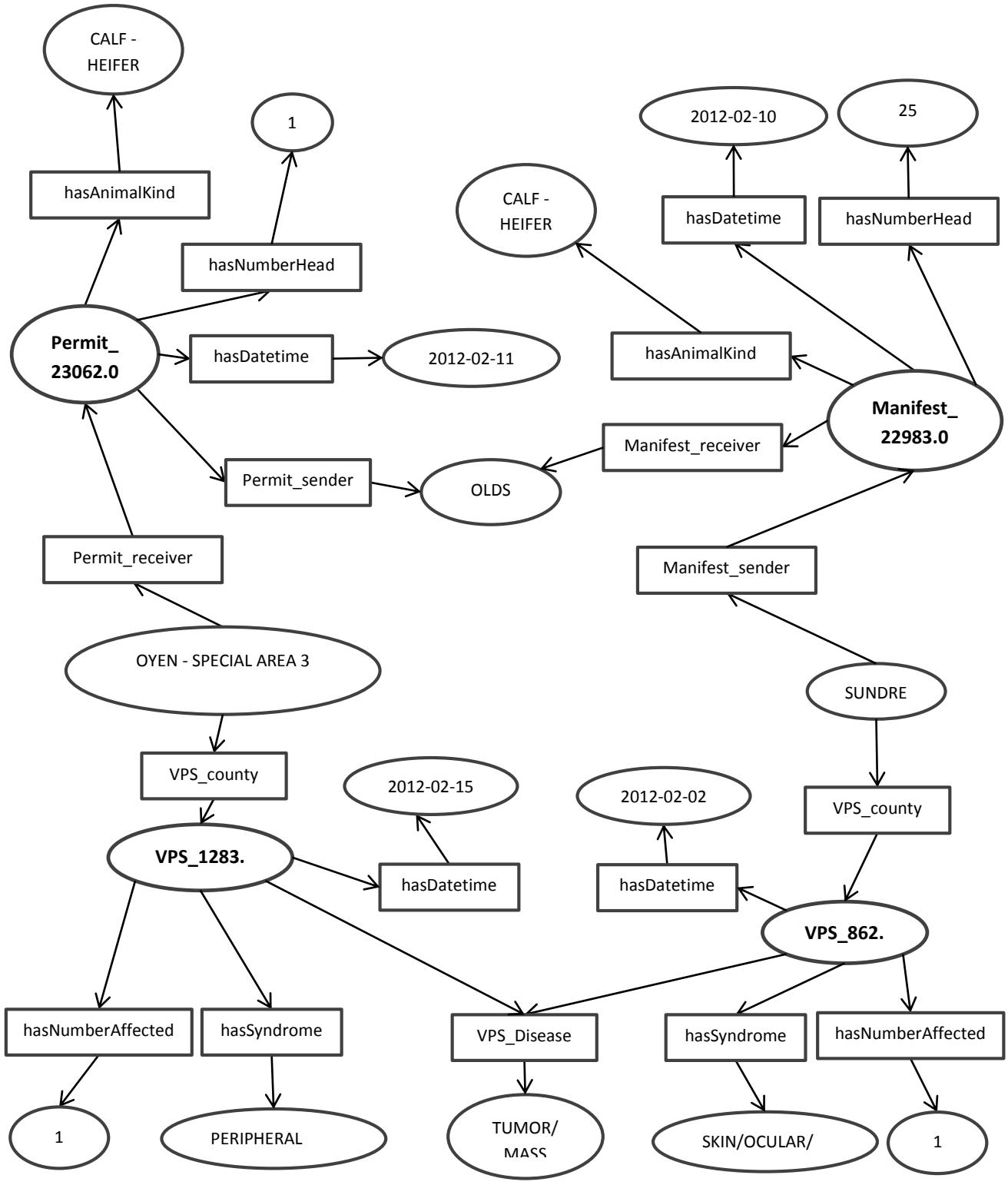


Figure 39 - Sample of RDF data

Fig. 39 shows a sample of RDF data representing a disease spreading from one county to another county. It is extracted from actual RDF data manually. As we can see there is reported "TUMOR/MASS" disease in "SUNDOR" county on "2012-02-02". Also, an animal movement from "SUNDOR" country to "OYEN – SPECIAL AREA 3" country occurred on "2012-02-10" and "2012-02-11", and an auction site "OLDS" was involved. Further, an animal disease was reported on "OYEN – SPECIAL AREA 3" county on "2012-02-15". The same disease occurred in "SUNDOR" county – and this creates a possibility that this disease was spreading within the province due to the movement of animals. This and similar examples of important movement/disease information can be found by writing a simple query using LORI.

Due to high level of complexity of this RDF dataset, an expert needs to create high level predicates for other users to use them during construction of their queries. These high level predicates make the users' queries simpler, as well as provide the users with the ability of merging the results of queries performed using LORI. Below, there is a list of high level predicates related to the agriculture data:

COUNTY_HAS_DISEASE (.*1) ->

(?MainSubject rdfs:VPS_county ?interconnectNode),

(?interconnectNode rdfs:VPS_Disease ?*1)

COUNTY_RECEIVED_FROM (.*1) ->

(?MainSubject rdfs:Permit_receiver ?interconnectNode),

(?interconnectNode rdfs:Permit_sender ?*1)

COUNTY_SENDED_TO (.*1) ->

(?MainSubject rdfs:Manifest_sender ?interconnectNode),

(?interconnectNode rdfs:Manifest_receiver ?*1)

COUNTY_RECEIVED_FROM_COUNTY (.*1, .*2, .*3) ->

(?MainSubject rdfs:Permit_receiver ?interconnectNode),

(?interconnectNode rdfs:Permit_sender ?auction),

(?interconnectNode rdfs:hasDatetime ?permitDatetime),

(?*1 rdfs:Manifest_sender ?interconnectNode1),

(?interconnectNode1 rdfs:Manifest_receiver ?auction),

approx_at_instant_before(?interconnectNode1,?*2,?*3,?permitDatetime,?tManifest),

greaterThan(?tManifest, 0.0)

The predicate *COUNTY_HAS_DISEASE(*1)* returns all counties that reported a disease. The parameter ".*1", which is optional, passes a disease name (instance) to the predicate for "filtering" the counties that reported a specific disease. The definition of that high level predicate includes two triples "connected" via a "relation node" (Section 4.1.2). Here we define a variable "?interconnectNode" to represent that node. That node can have any name – it does not affect the result because it is just a variable name that is used by the Jena inference system. The predicate *COUNTY_RECEIVED_FROM(*1)* returns all counties that have receives animals from auction places. The parameter ".*1" can be passed for filtering the results for a specific auction place. The *COUNTY_SENDED_TO(*1)* predicate returns all counties which send animals to auction places. And again, the parameter ".*1" can be passed to filter the result for a specific auction place. Please, notice that all parameters have to be of a specific RDF class as predicates use the parameters to filter the results. For the case of the last two predicates parameters should be auction instances. The last predicate: *COUNTY_RECEIVED_FROM_COUNTY(*1)* is the most complex one. It returns all counties that received animals from other counties through any auction places. The first parameter is a county instance that is used to filter sender counties. The second and the third parameters are related to temporal aspects of data and are used to define an approximate time of interest when a movement happened. The third parameter determines a moment in time, while the second indicates a number of time granules before that moment. The temporal predicate *approx_at_instant_before()* is used here, Section 4.3.5. It is used to filter the results and find the movements between counties that happened approximately in a specific period of time.

Queries prepared by the user are passed to the LORI Mapping Engine (Section 4.2.5) where all high level predicates are replaced by their definitions included in the Rule Mappings.

An example of the high level query that uses a number of high level predicates is shown below:

COUNTY_HAS_DISEASE(TRAUMA), approx_at_instant('2012-02-15',0) -> StoreResult(ResultTrauma)

The result is a list of counties that reported TRAUMA approximately on February 2nd of 2012. The list is saved under the name *ResultTrauma*, and can be used for merging or further processing. Below, there is the result of running this query against the agriculture RDF dataset. The results include execution times that are treated as performance measures:

```
+++++++++ Start loading rdf data +++++++++

++++++++++++++++++ End loading rdf data (6430 ms) ++++++

+++++++++ Start reasoning +++++++++

++++++++++++++++++ End reasoning (245 ms) ++++++++++++++

+++++++++ Start preparing infModel +++++++++++++++++
ResultTruma(1): KNEEHILL (TRAUMA ) ( 1.0 )
ResultTruma(2): NEWELL (TRAUMA ) ( 0.5 )
ResultTruma(3): BIRCH HILLS (TRAUMA ) ( 0.5 )
ResultTruma(4): PONOKA (TRAUMA ) ( 1.0 )
ResultTruma(5): TABER (TRAUMA ) ( 1.0 )
ResultTruma(6): LACOMBE (TRAUMA ) ( 1.0 )
ResultTruma(7): PONOKA (TRAUMA ) ( 1.0 )
ResultTruma(8): LEDUC (TRAUMA ) ( 1.0 )
ResultTruma(9): PONOKA (TRAUMA ) ( 0.5 )
++++++++++++ End preparing infModel (247 ms) +++++++
```

As it can be observed, there are nine reported “TRAUMA” disease cases that occurred approximately on February 2nd, 2012 in different counties. The numbers inside the brackets show the degree of closeness of the reported disease date to February 2nd, 2012. We can filter these results based on the degrees of closeness. This can be

accomplished using the following query that selects only cases with the closeness degree above 0.6:

COUNTY_HAS_DISEASE(TRAUMA), approx_at_instant('2012-02-15',0.6) -> StoreResult(ResultTruma)

The results are:

```
+++++++++ Start preparing infModel +++++++++++++++++
---> "ResultTruma"(1): KNEEHILL (TRAUMA ) ( 1.0 )
---> "ResultTruma"(2): PONOKA (TRAUMA ) ( 1.0 )
---> "ResultTruma"(3): TABER (TRAUMA ) ( 1.0 )
---> "ResultTruma"(4): LACOMBE (TRAUMA ) ( 1.0 )
---> "ResultTruma"(5): PONOKA (TRAUMA ) ( 1.0 )
---> "ResultTruma"(6): LEDUC (TRAUMA ) ( 1.0 )
+++++++++ End preparing infModel (296 ms) ++++++++++
```

The result shows reported animal disease cases that occurred "closer" (time-wise) to the requested date. Another query – below – is an example of an application of aggregation operation on the results. The query and its result are:

COUNTY_HAS_DISEASE(TRAUMA), approx_at_instant('2012-02-15',0.6) -> StoreResult(ResultTruma, MOST)

```
+++++++++ Start preparing infModel +++++++++++++++++++
---> "ResultTruma"(1): PONOKA (TRAUMA ) ( 0.5 ) ( 3.0 )
---> "ResultTruma"(2): BIRCH HILLS (TRAUMA ) ( 0.5 ) ( 1.0 )
---> "ResultTruma"(3): LACOMBE (TRAUMA ) ( 1.0 ) ( 1.0 )
---> "ResultTruma"(4): TABER (TRAUMA ) ( 1.0 ) ( 1.0 )
---> "ResultTruma"(5): NEWELL (TRAUMA ) ( 0.5 ) ( 1.0 )
---> "ResultTruma"(6): LEDUC (TRAUMA ) ( 1.0 ) ( 1.0 )
---> "ResultTruma"(7): KNEEHILL (TRAUMA ) ( 1.0 ) ( 1.0 )
++++++++++++ End preparing infModel (232 ms) +++++++++
```

The aggregation is done based on counties with "TRAUMA" disease cases. For example there are three cases of "TRAUMA" reported in "PONOKA" which are close to February 2nd of 2012.

The list of queries shown below is an example of a "data query session" that has a single objective of finding an event of spreading a specific disease from one county to another due to animal movement:

rule1: COUNTY_HAS_DISEASE(TRAUMA), approx_at_instant('2012-02-04',0) -> StoreResult(Result1)

rule2: COUNTY_SENDED_TO(LETHBRIDGE), approx_at_instant('2012-02-06',0) -> StoreResult(Result2)

rule3: COUNTY_RECEIVED_FROM(LETHBRIDGE), approx_at_instant('2012-02-11',0.5) -> StoreResult(Result3)

rule4: COUNTY_HAS_DISEASE(TRAUMA), approx_at_instant('2012-02-14',0) -> StoreResult(Result4)

rule5: MergeResults(Result1,Result2) -> StoreResult(Result5)

rule6: MergeResults(Result3,Result4) -> StoreResult(Result6)

rule7: MergeResults(Result5,Result6) -> StoreResult(Result7)

Let us analyze the queries. The first one finds counties that had reported "TRAUMA" approximately around February 4th of 2012. The second one finds counties that send animals to "LETHBRIDGE" auction place approximately around February 6th of 2012. The third rule finds counties that received animals from "LETHBRIDGE" auction place approximately around February 11th of 2012. The fourth rule finds counties that reported cases of "TRAUMA" disease approximately around February 14th of 2012. In order to determine the final result we merge the results obtained from individual queries. The first and second results are merged together to returns counties that reported "TRAUMA" and sent animals to "LETHBRIDGE" after that. The merge between the third and fourth results returns counties that received animals from "LETHBRIDGE" and had reported "TRAUMA" after that. After that, the last two results are merged together to return the final result, i.e., all counties that reported "TRAUMA" in a given time interval and received animals from other counties that reported cases of the same disease. So, this implies possibility of spreading "TRAUMA" among counties. These queries can be run without parameters to return all possible situations with different diseases and all auction places. The result of such a query is shown below:

```
+++++++++ Start preparing infModel +++++++++++++++++++

---> "Result5"(1): TABER (LETHBRIDGE / TRAUMA ) ( 0.5 )
```

```
---> "Result6"(1): RED DEER (TRAUMA / LETHBRIDGE ) ( 1.0 )
---> "Result7"(1): TABER (LETHBRIDGE / RED DEER ) ( 0.5 )
---> "Result7"(2): TABER (TRAUMA / RED DEER ) ( 0.5 )

+++++++++ End preparing infModel (116030 ms) +++++++++
```

The result implies that there is a possible spread of a disease from the "TABER" county to the "RED DEER" county through the "LETHBRIDGE" auction place approximately between February 6th and February 15th of 2012.

5.2.2. Performance Evaluation

To evaluate performance of LORI and show its advantages, we design and execute a number of experiments. The idea has been to compare the results of several queries on the same RDF dataset with and without the developed predicates.

To the best of our knowledge, there is no system or tool capable of handling temporal RDF data with any aspects of fuzziness. Therefore, we use a standard SPARQL query language. The comparison is performed according to the following procedure: we build queries using LORI predicates and then try to build and run equivalent – if possible – queries in SPARQL. The results obtained in both cases are compared. Such evaluation allows us to claim that our approach is able to handle a wide range of queries related to temporal and imprecise information. In the experiments, we use a SPARQL tool called Twinkle. It is a simple Java GUI interface providing an access to a SPARQL query engine.

We have defined 3 different cases. Each of them is related to a different scenario linked to a query built using LORI-based predicates. Then we tried to mimic the same query in SPARQL and run it using Twinkle. Finally, we compare the results obtained with and without LORI predicates. In some cases, it is easy to create queries in SPARQL that are equivalent to queries in LORI. But in other cases, it is not even possible to mimic LORI as SPARQL cannot handle temporal aspects in a fuzzy manner.

First Case:

In the first case, we build queries without considering temporal or fuzzy aspects of data. The goal of this case is to show that LORI can also be used as a simple querying tool. Following shows the query that we ran in LORI:

COUNTY_HAS_DISEASE() -> StoreResult(Result1)
ShowResult(Result1)

This query gives just a list of all counties that reported any animal disease. This is the simplest query we can make. The result obtained using the LORI query includes 1669 counties:

```
++++++++++ Start preparing infModel ++++++++++

---> "Result1"(1): WETASKIWIN (TRAUMA )( 1.0 )
---> "Result1"(2): LETHBRIDGE (TRAUMA )( 1.0 )
---> "Result1"(3): ROCKY VIEW (TRAUMA )( 1.0 )
---> "Result1"(4): LEDUC (TRAUMA )( 1.0 )
---> "Result1"(5): SADDLE HILLS (TRAUMA )( 1.0 )
                  …
                  …
                  …
---> "Result1"(1665): DRUMHELLER (PROLAPSED RECTUM)( 1.0 )
---> "Result1"(1666): LEDUC (CYSTIC OVARIAN DISEASE)( 1.0 )
---> "Result1"(1667): CLEARWATER (BRD COMPLEX)( 1.0 )
---> "Result1"(1668): NEWELL (HEMMORHAGE/TRAUMA)( 1.0 )
---> "Result1"(1669): CLEARWATER (BVD)( 1.0 )

++++++++++++++++++++ End preparing infModel (2381 ms) ++++++++++

++++++++++++++++++++ Result Count: 1669 ++++++++++
```

As the query does not contain any temporal clause, the fuzzy membership values are all "1.0" which can be considered as an example of the crisp result.

The equivalent SPARQL query, shown below, is run using Twinkle. The results are compared with the one obtained with the LORI query.

PREFIX base: <http://www.fuzzytemporal.cs/vps#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

```
SELECT * WHERE
{
  ?MainSubject rdfs:VPS_county ?interconnectNode .
  ?interconnectNode rdfs:VPS_Disease ?object
}
```

Also for the SPARQL query the results include 1669 triples, which are identical to the triples obtained using the LORI query. Some of the triples are:

MainSubject	**Object**
http://www.fuzzytemporal.cs/vps#WETASKIWIN	http://www.fuzzytemporal.cs/vps#TRAUMA
http://www.fuzzytemporal.cs/vps#LETHBRIDGE	http://www.fuzzytemporal.cs/vps# TRAUMA
http://www.fuzzytemporal.cs/vps#ROCKY VIEW	http://www.fuzzytemporal.cs/vps# TRAUMA
http://www.fuzzytemporal.cs/vps#LEDUC	http://www.fuzzytemporal.cs/vps# TRAUMA
http://www.fuzzytemporal.cs/vps#SADDLE HILLS	http://www.fuzzytemporal.cs/vps# TRAUMA
.	.
.	.
.	.
http://www.fuzzytemporal.cs/vps#*DRUMHELLER*	http://www.fuzzytemporal.cs/vps#*PROLAPSED RECTUM*
http://www.fuzzytemporal.cs/vps#*LEDUC*	http://www.fuzzytemporal.cs/vps#*CYSTIC OVARIAN DISEASE*
http://www.fuzzytemporal.cs/vps#*CLEARWATER*	http://www.fuzzytemporal.cs/vps#*BRD COMPLEX*
http://www.fuzzytemporal.cs/vps#*NEWELL*	http://www.fuzzytemporal.cs/vps#*HEMMORHAGE/TRAUMA*
http://www.fuzzytemporal.cs/vps#*CLEARWATER*	http://www.fuzzytemporal.cs/vps#*BVD*

Second Case:

In this case we have added, compared to the query from the previous case, a temporal clause. Furthermore, we have filtered the results for a specific animal disease. The goal of this example is to show that LORI can deal with temporal data in a fuzzy manner. The following shows the query we have run in LORI:

```
COUNTY_HAS_DISEASE(TRAUMA), approx_at_instant('2012-02-15',0) ->
StoreResult(Resul2)
```

The obtained results contain all counties that reported “TRAUMA” disease cases approximately on February 15nd, 2012. Also, the results give us an indication about

closeness of occurring these disease cases to the desired date included in the query. The LORI's results are shown below. They include 10 triples:

```
+++++++++ Start preparing infModel +++++++++

---> "Result2"(1): PONOKA (TRAUMA)( 0.5 )
---> "Result2"(2): TABER (TRAUMA)( 1.0 )
---> "Result2"(3): PONOKA (TRAUMA)( 1.0 )
---> "Result2"(4): LEDUC (TRAUMA)( 1.0 )
---> "Result2"(5): LACOMBE (TRAUMA)( 1.0 )
---> "Result2"(6): NEWELL (TRAUMA)( 0.5 )
---> "Result2"(7): BIRCH HILLS (TRAUMA)( 0.5 )
---> "Result2"(8): RED DEER (TRAUMA)( 1.0 )
---> "Result2"(9): PONOKA (TRAUMA)( 1.0 )
---> "Result2"(10): KNEEHILL (TRAUMA)( 1.0 )

++++++++++++++++++ End preparing infModel (257 ms) +++++++++

++++++++++++++++++ Result Count: 10 +++++++++
```

As we can see, some of the triples are closer to the desired date – their fuzzy membership values are "1.0", while some of them have the value "0.5" which means they are a bit further from the date of interest. In this case, we use a trapezoidal membership function with parameters: a = -4, b = -2, c = +2 and d = +4 which are: "a" is lower bound, "b" is min flat part, "c" is max flat part and "d" is upper bound, Fig. 40. Please notice that for this example we use a time granularity of "day". For example, when we query for approximately February 15nd, 2012 values for membership function parameters are: a= February 11nd, b= February 13nd, c= February 17nd, and d= February 19nd, 2012.

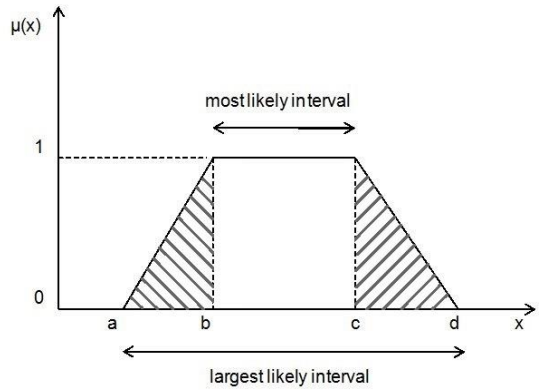


Figure 40 - Trapezoidal membership function with shaded fuzzy part

In this case of a SPARQL query, we have tried to mimic the LORI query but SPARQL does not have abilities to handle fuzzy temporal queries. In order to accomplish that, we have filtered the results for the cases between February 11nd and February 19nd, 2012 as LORI considers this period of time when it has looked at the data via a trapezoidal membership function. The SPARQL query is:

```
SELECT * WHERE
{
        ?MainSubject rdfs:VPS_county ?interconnectNode .
        ?interconnectNode rdfs:VPS_Disease <http://www.fuzzytemporal.cs/vps#TRAUMA> .
        ?interconnectNode rdfs:hasDatetime ?datetime .
       FILTER (?datetime > '2012-02-11') .
       FILTER (?datetime < '2012-02-19')
}
```

As the result we have obtained 10 RDF triples that is the exact number of triples as for the query from the first case without a temporal component.

MainSubject	Datetime
http://www.fuzzytemporal.cs/vps# LEDUC	2012-02-13
http://www.fuzzytemporal.cs/vps# LACOMBE	2012-02-17
http://www.fuzzytemporal.cs/vps# RED DEER	2012-02-14
http://www.fuzzytemporal.cs/vps# NEWELL	2012-02-18
http://www.fuzzytemporal.cs/vps# TABER	2012-02-15
http://www.fuzzytemporal.cs/vps# PONOKA	2012-02-17
http://www.fuzzytemporal.cs/vps# KNEEHILL	2012-02-17
http://www.fuzzytemporal.cs/vps# BIRCH HILLS	2012-02-12
http://www.fuzzytemporal.cs/vps# PONOKA	2012-02-13
http://www.fuzzytemporal.cs/vps# PONOKA	2012-02-18

The results of LORI and Twinkle are exactly the same in terms of numbers of triples. The difference is that the SPARQL query is not able to handle temporal fuzzy concepts and all results are treated in the same way – all equally important. LORI, on the other hand, provides values representing degrees of closeness of triples to the desire dates. The results that "fit" the shaded part of Fig. 40 satisfy the query requirement to a degree. For example, for the disease reported in the "NEWELL" county that occurred on February 18nd, 2012 LORI determines its importance as lower than for the disease reported in the "LEDUC" county. If we change search criteria in the SPARQL query to accommodate a narrower time window, the obtained results will include a lower number of responses. This could involve a loss of important results. Below, we show a new SPARQL query with new boundaries, together with the results:

```
SELECT * WHERE
{
 ?MainSubject rdfs:VPS_county ?interconnectNode .
 ?interconnectNode rdfs:VPS_Disease <http://www.fuzzytemporal.cs/vps#TRAUMA> .
 ?interconnectNode rdfs:hasDatetime ?datetime .
FILTER (?datetime > '2012-02-13') .
FILTER (?datetime < '2012-02-17')
 }
```

MainSubject	Datetime
http://www.fuzzytemporal.cs/vps# RED DEER	2012-02-14
http://www.fuzzytemporal.cs/vps# TABER	2012-02-15

Third Case:

In this case, we build a complex query in LORI that involves multiple rules related to temporal aspects of data. The query looks as follow:

Rule1: COUNTY_HAS_DISEASE(TRAUMA), approx_at_instant('2012-02-04',0) -> StoreResult(testResult1)

Rule2: COUNTY_SENDED_TO(LETHBRIDGE), approx_at_instant('2012-02-06',0) -> StoreResult(testResult2)

Rule3: COUNTY_RECEIVED_FROM(LETHBRIDGE), approx_at_instant('2012-02-11',0) -> StoreResult(testResult3)

Rule4: COUNTY_HAS_DISEASE(TRAUMA), approx_at_instant('2012-02-14',0) -> StoreResult(testResult4)

Rule5: MergeResults(testResult1,testResult2) -> StoreResult(testResult5)

Rule6: MergeResults(testResult3,testResult4) -> StoreResult(testResult6)

Rule7: MergeOutterResults(testResult5,testResult6) -> StoreResult(testResult7)

Rule8: ShowResultDistinct(testResult7)

The obtained results are:

```
+++++++++ Start preparing infModel +++++++++++++++++++

---> "Result7"(1): TABER (LETHBRIDGE / RED DEER ) ( 0.5 )
---> "Result7"(2): TABER (TRAUMA / RED DEER ) ( 0.5 )

+++++++++ End preparing infModel (107579 ms) +++++++++

++++++++++++++++++ Result Count: 2 +++++++++
```

They show the spread of disease from the county “TABER” to the county “RED DEER”. In this case, we have two choices for writing a SPARQL query to mimic the above LORI query.

The first option is to create two sets of queries, execute each of them separately and merge the results manually. The first set of SPARQL queries would provide results similar to the results obtained from rules 1 and 2 of the LORI query (above), and the second set includes SPARQL queries related to LORI’s rules 3 and 4. The second option is to run both sets of queries at the same time. The first option is faster but merging of results is difficult. The second option gives us the final results but it is slow. Also, we need to further alter the SPARQL query via adding a keyword “Distinct”. This would remove any redundant results; otherwise the result has lot of unneeded RDF triples. The large SPARQL query representing the second option above is:

```
SELECT DISTINCT ? MainSubject  ? MainSubject2 WHERE
{
   ?MainSubject rdfs:VPS_county ?interconnectNode .
   ?interconnectNode rdfs:VPS_Disease <http://www.fuzzytemporal.cs/vps#TRAUMA> .
   ?interconnectNode rdfs:hasDatetime ?datetime .
         FILTER (?datetime > '2012-01-31') .
         FILTER (?datetime < '2012-02-08') .

   ?MainSubject rdfs:Manifest_sender ?interconnectNode2 .
   ?interconnectNode2 rdfs:Manifest_receiver <http://www.fuzzytemporal.cs/vps#LETHBRIDGE>
.
   ?interconnectNode2 rdfs:hasDatetime ?datetime2 .
         FILTER (?datetime2 > '2012-02-02') .
         FILTER (?datetime2 < '2012-02-10') .

   ?MainSubject2 rdfs:Permit_receiver ?interconnectNode3 .
   ?interconnectNode3 rdfs:Permit_sender <http://www.fuzzytemporal.cs/vps#LETHBRIDGE> .
   ?interconnectNode3 rdfs:hasDatetime ?datetime3 .
         FILTER (?datetime3 > '2012-02-07') .
         FILTER (?datetime3 < '2012-02-15')

   ?MainSubject2 rdfs:VPS_county ?interconnectNode4 .
   ?interconnectNode4 rdfs:VPS_Disease <http://www.fuzzytemporal.cs/vps#TRAUMA> .
   ?interconnectNode4 rdfs:hasDatetime ?datetime4 .
         FILTER (?datetime4 > '2012-02-10') .
         FILTER (?datetime4 < '2012-02-18')
}
```

The result of the above SPARQL query is depicted below, and it is exactly the same as LORI result.

MainSubject	**MainSubject2**
http://www.fuzzytemporal.cs/vps# TABER	http://www.fuzzytemporal.cs/vps# RED DEER

Again the difference is that the SPARQL query is not able to handle temporal fuzzy data. The LORI result indicates that the degrees of closeness for all reported animal disease cases and movements are “0.5” or more. So, if we just change the date boundaries in SPARQL query, see the query below, an empty dataset is returned. It is because SPARQL query looks at data in “a crisp manner”.

```
SELECT DISTINCT ? MainSubject  ? MainSubject2 WHERE
{
   ?MainSubject rdfs:VPS_county ?interconnectNode .
   ?interconnectNode rdfs:VPS_Disease <http://www.fuzzytemporal.cs/vps#TRAUMA> .
```

```
?interconnectNode rdfs:hasDatetime ?datetime .
        FILTER (?datetime > '2012-02-02') .
        FILTER (?datetime < '2012-02-06') .

?MainSubject rdfs:Manifest_sender ?interconnectNode2 .
?interconnectNode2 rdfs:Manifest_receiver <http://www.fuzzytemporal.cs/vps#LETHBRIDGE>
.
?interconnectNode2 rdfs:hasDatetime ?datetime2 .
        FILTER (?datetime2 > '2012-02-04') .
        FILTER (?datetime2 < '2012-02-8') .

?MainSubject2 rdfs:Permit_receiver ?interconnectNode3 .
?interconnectNode3 rdfs:Permit_sender <http://www.fuzzytemporal.cs/vps#LETHBRIDGE> .
?interconnectNode3 rdfs:hasDatetime ?datetime3 .
        FILTER (?datetime3 > '2012-02-09') .
        FILTER (?datetime3 < '2012-02-13')

?MainSubject2 rdfs:VPS_county ?interconnectNode4 .
?interconnectNode4 rdfs:VPS_Disease <http://www.fuzzytemporal.cs/vps#TRAUMA> .
?interconnectNode4 rdfs:hasDatetime ?datetime4 .
        FILTER (?datetime4 > '2012-02-12') .
        FILTER (?datetime4 < '2012-02-16')
}
```

5.3. LORI Performance

One of important aspects of LORI's implementation has been its performance. This is especially important in the context of large volume data sets. Many development concerns – also related to a selection of RDF format for N-ary relations (Section 4.1.1) – has been driven by a need to ensure a good inference performance of LORI. The idea of "relation nodes" ensures that RDF binary relations are the only way of representing any type of information using RDF data. Another source of possible performance degradation is utilization of fuzzy temporal predicates. For each high-level rule/query that contains one of fuzzy temporal predicates, Jena inference system calls a function associated with the predicate that executes this predicate and evaluates its result. That affects the performance of running rules/queries on RDF data. However, there is no choice but to accept this overhead in order to gain the ability to build RDF queries with fuzzy terms, and to work with temporal data in a complex RDF structure.

We have conducted a performance experiment on the agriculture RDF data in order to identifying that overhead. In this experiment we have run regular LORI queries ten times, and have logged all elapsed times required to obtain the results. In the next step, we have removed all LORI predicates and replaced them with regular RDF triples, have run such queries also ten times and saved all elapsed times. Obviously some LORI predicates like fuzzy temporal predicates cannot be replaced with anything from RDF reasoner rules; so we have missed some functionality in these situations.

The first experiments have been conducted with the query about the animal disease "TRAUMA" reported approximately on February 15nd, 2012. Following is the query with LORI predicates:

(?MainSubject rdfs:VPS_county ?interconnectNode),
(?interconnectNode rdfs:VPS_Disease http://www.fuzzytemporal.cs/vps#TRAUMA),
approx_at_instant(?interconnectNode, '2012-02-15', ?tMain), greaterThan(?tMain, 0.0)
->
StoreResult('result1',?MainSubject,null,?tMain,?http://www.fuzzytemporal.cs/vps#TRAUMA)

A similar query without LORI predicates is:

?MainSubject rdfs:VPS_county ?interconnectNode),
(?interconnectNode rdfs:VPS_Disease http://www.fuzzytemporal.cs/vps#TRAUMA),
-> (?MainSubject rdfs:hasNode http://www.fuzzytemporal.cs/vps#TRAUMA)

Here, we have removed temporal predicates, and also have made changes regarding a way the results are stored: instead of storing them under a name, we have just created a new relation between the main subject, which is a county, and a specific disease. The result of running these queries on the agriculture RDF dataset is shown in Table 5:

Table 5 - Result of performance in case one

Iteration	Query With LORI Predicates (ms)	Query Without LORI Predicates (ms)
1	211	129
2	214	131
3	219	136
4	216	144
5	216	128
6	214	139
7	203	172
8	218	167
9	203	153
10	203	163
Average (ms)	**211**	**146**
STDEV (ms)	**6.4**	**16.4**

Based on these results, we conclude that the overhead introduced by LORI predicates is about 64 milliseconds or 44 per cent. Because the query is simple and LORI temporal predicates are given explicit times, so it looks that the overhead is a bit extensive. Therefore, more experiments have been designed and conducted.

The second set of experiments has included the query about animal movements from counties to counties through the auction place "LETHBRIDGE", when a movement to the auction place happened approximately on February 2nd, 2012, while a movement from the auction place happened approximately 4 days after moving to the auction place. The query with LORI predicates is:

(?MainSubject rdfs:Permit_receiver ?interconnectNode),
(?interconnectNode rdfs:Permit_sender http://www.fuzzytemporal.cs/vps#LETHBRIDGE),
(?interconnectNode rdfs:hasDatetime ?permitDatetime),
(http://www.fuzzytemporal.cs/vps#DUCHESS rdfs:Manifest_sender ?interconnectNode1),

(?interconnectNode1 rdfs:Manifest_receiver http://www.fuzzytemporal.cs/vps#LETHBRIDGE),
approx_at_instant_before(?interconnectNode1,4,DAYS,?permitDatetime,?tManifest),
greaterThan(?tManifest, 0.0),
approx_at_instant(?interconnectNode, '2012-02-12',?tMain),
greaterThan(?tMain, 0.0)
-> StoreResult('Result1',?MainSubject,null,?tMain,http://www.fuzzytemporal.cs/vps#DUCHESS)

And the query without LORI predicates is as follows:

(?MainSubject rdfs:Permit_receiver ?interconnectNode),
(?interconnectNode rdfs:Permit_sender http://www.fuzzytemporal.cs/vps#LETHBRIDGE),
(?interconnectNode rdfs:hasDatetime ?permitDatetime),
(http://www.fuzzytemporal.cs/vps#DUCHESS rdfs:Manifest_sender ?interconnectNode1),
(?interconnectNode1 rdfs:Manifest_receiver http://www.fuzzytemporal.cs/vps#LETHBRIDGE)
-> (?MainSubject rdfs:hasNode http://www.fuzzytemporal.cs/vps#TRAUMA)

Here, we have removed all LORI predicates and instead of storing the result under a specific name we have just created a new relation between the main subject that is a county and a specific disease. The result of running these queries on the agriculture dataset is shown in Table 6.

Table 6 - Result of performance in case two

Iteration	Query With LORI Predicates (ms)	Query Without LORI Predicates (ms)
1	643,139	646,968
2	661,186	642,655
3	619,545	623,589
4	635,628	639,042
5	611,328	615,988
6	619,949	618,844
7	644,192	624,026
8	642,846	638,345
9	637,463	620,858
10	639,082	634,255
Average (ms)	**635,436**	**630,457**
STDEV (ms)	**14,700.6**	**11,038.5**

In this case, the overhead of using LORI predicates is about 4,979 milliseconds or 0.78 percent. The fact that it is less than 1 percent makes it very reasonable to ignore it. This means that utilization of LORI predicates in complex queries does not influence the performance of LORI.

6. Conclusions and Future Work

The introduction of the Semantic Web has brought a data representation format – Resource Description Framework (RDF) – suitable for expressing relations between individual pieces of information. RDF is a fundamental format of data representation used in Lined Open Data. It is also applied for representing "enhanced" taxonomy – ontology. RDF is an encouraging step towards creating a foundation for methods and approaches providing a different, more intelligent and human-oriented way of processing, analyzing and utilization of data and information.

There is a growing demand for representing more sophisticated data involving multiple features, relations, and temporal aspects. Quite often, such data contain time related information expressed in an approximate manner. This is particularly evident in the case of data provided by humans – these data contain vague terms describing variety of facts.

6.1. Contributions

This thesis presents a fuzzy-based methodology suitable for building a system that provides the user with the ability to exploit temporal data. This exploitation happens without an extensive knowledge of details related to the structure of queried data, and can be performed using imprecise expressions built with quantitative and time-based terms. The concepts required for developing a system for interacting with RDF temporal data are presented.

The two chapters of the thesis – Chapter 3: Fuzzy Temporal Data in Ontology Environment and Chapter 4: LORI Linguistically-Oriented RDF Interface – describe the main contributions of the thesis. They address the above-mentioned issues with OWL and RDF as the data representation formats. Overall, a detailed list of contributions includes:

- designing and implementation of ontologies capable of representing temporal as well as fuzzy information;
- developing temporal predicates as built-in functions of Jena's that can be utilized as atoms during a process of constructing ontology-based IF-THEN rules in SWRL (Semantic Web Rule Language);
- designing an architecture of LORI based on an idea of two interfaces: 1) low-level one called *ReasonerInterface* which provides necessary rules and predicates to deal with temporal and complex data, built on Jena's RDF/RDFS reasoner; and 2) *UserInterface* composed of high-level predicates and built by data expert based on *ReasonerInterface*;
- developing temporal predicates as built-in functions of Jena's RDF/RDFS reasoner; these predicates utilize fuzzy terms to express imprecise declarations of time;
- proposing and developing a data structure for storing query results, with the ability to create sequences of queries, as well as to store, process and merge individual results;
- identifying a flexible approach for mapping high-level queries to low-level ones; the proposed idea is based on a special mapping file that allows for dynamic changes and modifications of mapping rules.

The mentioned contributions are the initial steps towards building a comprehensive methodology suitable for storing and utilizing RDF data with temporal information and N-ary relations.

Although the research has reached its aims, there are still some limitations of the proposed approach.

- In LORI, we proposed an RDF blank node for adding more information about a property linking two concepts. This RDF blank node is required for hiding all complex RDF data structures and predicates. When the processed data does not use blank nodes, users have to have some basic knowledge about used RDF schema and temporal properties. At the same time, the users have to know names

of temporal properties to use them for constructing queries with temporal predicates.

- LORI has the capability of using different fuzzy membership functions for fuzzy temporal predicates. Currently, these membership functions are configurable at the level of LORI source code. An interface for defining and setting parameters' values of different fuzzy membership functions is needed. Expert users could use it for customization purposes.
- The performance of LORI when complex fuzzy membership functions are used degrades substantially when compared with the performance when simple functions are used. For example, Gaussian fuzzy membership functions have high complexity, and their application requires improvements of the LORI's implementation.

6.2. Future Work

The presented methodology for handling fuzzy temporal data represented using OWL (ontology) and RDF data representation formats can be treated a starting point for more advanced and focused research tasks. Some of possible new research activities include:

- designing and developing a friendly interface for experts to build high-level predicates, and to prepare files with mapping rules;
- extending selection of fuzzy predicates suitable for dealing with multiple types of information attached to different data nodes;
- developing and implementing fuzzy rule-based reasoning engines based on Mamdani and Takag-Sugeno models;
- designing and developing predicates that work directly on RDF data and take advantage of its graph-based format; for example, predicates with functions for estimating similarity between entities, or predicates with graph analysis functions

for finding connections between entities mapped into a problem of a finding a short path in graphs;

- adding an interface for graphical-based visualization and analysis of RDF data before and after execution of predicates.

Bibliography

[1] A. Artale, and E. Franconi, "Temporal desc ription logics", in: Handbook of Time and Temporal Reasoning in Artificial Intelligence. Elsevier, Amsterdam, 2005.

[2] A. Frigeri, L. Pasquale, P. Spoletini, "Fuzzy Time in LTL". ArXiv: 1203.6278 v1., March 2009.

[3] A. Pugliese, O. Udrea, and V.S. Subrahmanian, "Scaling RDF with Time", Proc. Second European Semantic Web Conf. (ESWC '05), 2005, pp. 93-107.

[4] C. Bizer, T. Heath and T. Berners-Lee, "Linked Data – The Story So Far". *International Journal on Semantic Web and Information Systems*, Vol. 5, No. 3, pp.1-22. DOI: 10.4018/jswis.2009081901.

[5] C. Gutierrez, C. Hurtado, and A. Vaisman, "Introducing Time into RDF", IEEE Transaction on Knowledge And Data Engineering (ESWC '05), 19(2), 2007, pp. 207-218.

[6] C. Gutierrez, C. Hurtado, and A. Vaisman, "Temporal RDF," Proc. Second European Semantic Web Conf. (ESWC '05), 2005, pp. 93-107.

[7] D. Dubois, H. Prade, "Processing fuzzy temporal knowledge*",* IEEE Trans. on Systems Man& Cybernetics 19, 1989, pp. 729-744.

[8] F. Pan, and J.R. Hobbs, "Time in OWL-S", AAAI Spring Symposium on Semantic Web Services, Stanford University, CA, 2004, pp. 29-36.

[9] G. Antoniou , F. van Harmelen, "A Semantic Web Primer", *MIT Press*, Cambridge, MA, 2004.

[10] H. Kim, and K. Oh, "A new representation model in uncertain temporal knowledge", in: Proc. IFSA'91, 1991, pp. 113-116.

[11] Heath, T., & Bizer, C. (2011). "Linked Data: Evolving the Web into a Global Data Space". *Synthesis Lectures on the Semantic Web: Theory and Technology*, 1:1, 1-136. Morgan & Claypool.

[12] Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M., SWRL, "A Semantic Web rule language combining OWL and RuleML". W3C, 2004.

[13] http://blog.iandavis.com/2009/08/representing-time-in-rdf-part-1 (accessed July 9th, 2015)

[14] http://jena.apache.org/ (accessed July 9th, 2015)

[15] http://jena.apache.org/documentation/inference/#rdfs (accessed July 9th, 2015)

[16] http://linkeddata.org (accessed July 9th, 2015)

[17] http://lod-cloud.net/#history (accessed July 9th, 2015)

[18] http://lov.okfn.org/dataset/lov/ (accessed July 9th, 2015)

[19] http://rorchard.github.io/FuzzyJ/ (accessed July 9th, 2015)

[20] http://www.w3.org/standards/techs/rdfvocabs#w3c_all (accessed July 9th, 2015)

[21] http://www.w3.org/TR/ xmlschema11-1/ (accessed July 9th, 2015)

[22] http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140225/ (accessed July 9th, 2015)

[23] http://www.w3.org/TR/owl-features/ (accessed July 9th, 2015)

[24] http://www.w3.org/TR/rdf-schema/ (accessed July 9th, 2015)

[25] http://www.w3.org/TR/rdf-sparql-query/ (accessed July 9th, 2015)

[26] http://www.w3.org/TR/swbp-n-aryRelations/ (accessed July 9th, 2015)

[27] I. Horrocks, and P. F. Patel-Schneider, "Proposal for OWL Rule Language". *13th International World Wide Web Conference*, 2004, pp.723–731.

[28] J. Tappolet, and A. Bernstein, "Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL", in: Aroyo et al. The Semantic Web: Research and Applications, Lecture Notes in Computer Science Volume 5554, 2009, pp. 308-322.

[29] J.F. Allen, "A General Model of Action and Time", *Artificial Intelligence* Vol. 23, No. 2, 1984, pp. 123-154.

[30] J.F. Allen, "Maintaining Knowledge about Temporal Intervals", Communications of the ACM 26(11), 1983, pp. 832-843.

[31] L. Deng, Y. Yunpeng, C. Wang, Y. Jiang, "Fuzzy Temporal Logic on Fuzzy Temporal Constraint Network". Sixth International Conference on Fuzzy Systems and Knowledge Discovery. Tianjin, China, Aug. 14–16, 2009.

[32] M.J. O'Connor, and A.K. Das, "A Method for Representing and Querying Temporal Information in OWL". *ACM Transactions on Asian Language Processing* (TALIP): Special issue on Temporal Information Processing, 3(1): 2004, pp.66-85.

[33] Michael, H., Wolfgang, H., Yves, R., Lee, F. SCOVO, "Using Statistics on the Web of Data". Lecture Notes In Computer Science, 2009, pp.708-722.

[34] N. F.Noyand D. L. McGuinness, "Ontology Development 101: A Guide to Creating Your First Ontology". Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Reports SMI-2001-0880, March 2001.

[35] P. F. Patel-Schneider, P. Hayes, and I. Horrocks, "OWL Web Ontology Language Semantics and Abstract Syntax". W3C Recommendation 10 February 2004, Available at http://www.w3.org/TR/2004/REC-owl-semantics-20040210/.

[36] Carinena, A.Bugarin, M.Mucientes, "A language for expressing expert knowledge using fuzzy temporal rules", EUSFLAT-ESTYLF Joint Conference, 1999, pp. 171-174.

[37] S. Barro, R. Marin, and J. Mira, "A model and a language for the fuzzy representation and handling of time", Fuzzy Sets and Systems, 61(2), 1994, pp. 153-174.

[38] S. Batsakis, E. Petrakis, "Temporal Representation and Reasoning in OWL", *Semantic Web at IOS press.*

[39] S. Dutta, "An event base fuzzy temporal logic", in: Proc. 18th IEEE Int, Symp. On Multiple-Valued Logic (Palma de Mallorca), 1988, pp. 64-71.

[40] S. Schockaert, M. D. Cock, E. E. Kerre, "Reasoning About Fuzzy Temporal Information from The Web: Towards Retrieval of Historical Events". *Soft Computing*, Vol. 14, No. 8, 2010, pp.869-886.

[41] S.-I. Moon and K.H. Lee, "Fuzzy linear temporal logic". *Proc. 2nd Int. Symp. Adv. Intell. Syst. (ISAIS)*, Daejeon, Korea, Aug. 24–25, 2001, P. 184–188.

[42] S.-I. Moon, k. H. Lee, D. Lee, "Fuzzy Branching Temporal Logic". *IEEE Transactions on Systems, man, And Cybernetic- Part B: Cybernetics*. Vol. 34, No. 2, P.1045-1055, 2004.

[43] T. Berners-Lee, and M. Fischetti, "Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web". *IEEE Transactions Professional Communication*, VOL.43, NO. 2, June 2000.

[44] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web". *Scientific American*. 284, 2001, pp. 34–43.

[45] T. E. Gruber, "A translation approach to portable ontology specifications", *Knowledge Acquisition*, 5, 1993, pp. 199-220.

[46] W. Pedrycz, and F. Gomide, Fuzzy Systems Engineering, "Toward Human-Centric Computing", *Wiley-IEEE Press*, 2007.

[47] Welty, C.A., Fikes, R., "A reusable ontology for fluents in OWL". Proceeding of the 4th Int'l. Conference on Formal Ontology In Information Systems. P.226-236, 2006.

[48] Z. Lu, J. Liu, J. C. Augusto, H. Wang, "A Many-Values Temporal Logic and Reasoning Framework for Decision Making". *Computational Intelligence in Complex Decision System*, 2010, pp.125-146.