

Technological Educational Institute of Crete
School of Applied Technology
Department of Informatics Engineering

Thesis

Design and Implementation of a Time Reasoner for Knowledge
Representation on RDFS



Nikolaidis Vasileios *AM: 2617*

Supervisor Professor: Papadakis Nikos

Heraklion – June 2014

Ευχαριστίες

Σε αυτό το σημείο θα ήθελα να ευχαριστήσω την οικογένεια μου για την αμέριστη συμπαράσταση που μου έδειξαν και την πίστη τους σε μένα καθ' όλη τη διάρκεια των σπουδών μου έως και το τέλος της παρούσας εργασίας.

Επίσης θα ήθελα να ευχαριστήσω τη φίλη μου Άννα, που με στήριξε ως το τέλος της εργασίας αυτής. Τέλος θα ήθελα να ευχαριστήσω τον καθηγητή κ. Νίκο Παπαδάκη, για την εμπιστοσύνη που έδειξε με το θέμα της συγκεκριμένης εργασίας.

Synopsis

Σχεδιασμός και υλοποίηση ενός χρονικού reasoner για εξαγωγή γνώσης σε RDFS έγγραφα

Διανύοντας την τρίτη δεκαετία από τη γέννηση του Παγκόσμιου Ιστού, οι χρήστες του οποίου πλέον απαριθμούν σε δισεκατομμύρια, οι ανάγκες για επικοινωνία, μάθηση και επιχειρηματικότητα εξελίσσεται ακόμη ραγδαία. Έτσι ο παγκόσμιος ιστός έχει την ανάγκη να εξελίσσεται σύμφωνα με τις ανάγκες της ανθρωπότητας. Κατά την πρώτη δεκαετία το web ήταν στατικό, αποτελούμενο από ιστοσελίδες κυρίως κειμένου με ελάχιστες δυνατότητες αναζήτησης πληροφορίας, αυτό άλλαξε με τον ερχομό των μηχανών αναζήτησης και της αρχής της εποχής της διαδραστικότητας, με το δυναμικό πλέον web 2.0. Σήμερα, ο ιστός αποτελείται από 980 εκατομμύρια ιστοσελίδες που περιέχουν κάθε είδους πληροφορία σε οποιαδήποτε μορφή.

Αυτό οδηγεί στη δημιουργία του επόμενου βήματος στην εξέλιξη του παγκόσμιου ιστού, το λεγόμενο σημασιολογικό ιστό ή web 3.0. Τον όρο αυτό μας τον έδωσε ο εφευρέτης του αρχικού παγκόσμιου ιστού, ο Tim Berners-Lee.

Σκοπός της παρούσας εργασίας είναι η δημιουργία ενός χρονικού reasoner, μιας μηχανής με τη δυνατότητα να εξάγει λογικά συμπεράσματα με τη χρήση κανόνων γραμμένων στην κατηγορηματική λογική. Ο χρονικός reasoner διαφέρει από ένα απλό reasoner, επειδή οι κανόνες που περιέχει αφορούν χρονικές καταστάσεις. Τους reasoner τους χρησιμοποιούμε σε έγγραφα μεταδεδομένων του σημασιολογικού ιστού. Τα έγγραφα αυτά μπορεί να είναι τύπου RDFS, Turtle, OWL κα.

Abstract

Nowadays, in the third decade of the birth of the Web, its users list in the billions and the need for communication, learning and entrepreneurship is still evolving rapidly. Thus the Web has the need to evolve according to the needs of humanity. During its first decade the web was static, consisting of mainly text sites with poor information search, this changed with the advent of search engines and the beginning of the era of lifelong activity, with the most dynamic web 2.0.

These days, the web consists of 980 million web pages containing all kind of information in any form. This leads to the creation of the next step in the evolution of the Web, called semantic web or web 3.0. This term was given to us from the inventor of the original World Wide Web, Tim Berners-Lee.

The purpose of this work is to create a time reasoner, a piece of code with the ability to draw inferences using rules written in predicate logic. The time reasoner differs from a simple reasoner, because the rules contain statements relating to time. The reasoner uses metadata from documents from the semantic web. Those documents may be of RDFS, Turtle, and OWL syntax.

Keywords: Semantic reasoner, time reasoner, RDFS, Allen's Integral Algebra, Semantic Web, Predicate Logic, Ontology, fluents, Jena API, Eclipse, Owl-Time

Contents

Design and Implementation of a Time Reasoner for Knowledge Representation on RDFS...	1
1 Web 3.0 and Web standards.....	6
1.1 World Wide Web (World Wide Web)	6
1.2 Semantic Web	6
1.3 Ontologies	6
1.4 XML.....	7
1.5 RDF.....	8
1.6 RDFS.....	8
1.7 Protégé	12
2 Artificial Intelligence.....	13
2.1 Artificial Intelligence and Logic	13
2.2 Knowledge Representation	13
2.3 Logic	14
2.4 Propositional Logic	15
2.5 Predicate Logic (First Order Logic).....	16
3 Fluents and Reasoners.....	19
3.1 Reasoner.....	19
3.2 Situation Logic.....	19
3.3 Fluents.....	20
3.4 Allen's Integral Algebra.....	21
4 Jena	25
4.1 Jena API.....	25
4.2 Inference Engines.....	30
4.3 SparQL.....	38
5 Methodology.....	40
6 Conclusions.....	46

1 Web 3.0 and Web standards

1.1 World Wide Web (World Wide Web)

The World Wide Web (commonly known as the Web) is a system of interlinked hypertext documents that are accessed through the Internet. Those documents are called Web Pages. To view those Pages you need a piece of software called a browser. Nowadays almost every computer, cellphone and tablet can access the Internet, bringing information to the people wherever they are. Information that consist of text, images, videos and other multimedia.

The invention of the Web is acclaimed from a British computer scientist and former CERN employee, Tim Berners-Lee.

1.2 Semantic Web



Semantic Web (or Web 3.0) is the third stage in the evolution of the Web (World Wide Web) in which the content of web pages will bring a predetermined structure based on metadata. This change is requested to assist the better kind of compilation and processing of information, forming sites and extraction of knowledge from them.

The Semantic Web uses existed technologies such as XML and URI as well as Web 2.0, but also grows as new as RDF, RDFS, OWL, SPARQL, etc. In the Semantic Web ontologies are predefined and developed by companies, research centers and organizations in order to better organize the data so that it is easier for search engines to find the information that the user wants and can extract knowledge from more complicated questions.

The term Semantic Web has been proposed by Tim Berners-Lee, inventor of the World Wide Web since 2001. In 2006 it has been adopted by the W3C (World Wide Web Consortium). The main goal is the representation of knowledge by computers. To make this possible there should be a mechanism of information processing by the rules of logic in order to draw conclusions, the creation of new knowledge, decision support or even to automatically perform actions.

1.3 Ontologies

Ontology is a formal and explicit definition of common and agreed conceptual formatting on a field of interest. This formal representation of knowledge as a set of concepts, relationships and properties can be used for reasoning (inference / new knowledge) and knowledge structured description of a field of interest. Ontologies are introduced as a structured framework for organizing information and are used mainly in Artificial Intelligence in the Semantic Web, in Bioinformatics, Library science and other disciplines / branches as a form of knowledge representation about the world. The most widely used free software for creating ontologies is Protégé and it's been developed by Stanford University.

Ontology issue

One problem that appears in the representation of ontologies is the time variable (temporal) information. The use of languages that use descriptive logic and binary relations such as RDF and OWL are not enough to solve the problem. What we need now is triadic relationships where the third argument is time. Suppose we have an ontology of objects and relationships that show the lives of residents of a town in the example below.

Ex: Vasilis lives in Riga Fereou Street 6

- object (Riga Fereou type Street)
- object (Vasilis type Resident)
- Relationship (livesIn Riga Fereou)

In this representation of the data the main drawback is the lack of the variable time. It is a contemporary (synchronic) data representation, while in the real world relationships between objects are timeless (diachronic) that change over time. The problem would satisfy the following embodiment.

Ex: Vasilis lives in Riga Fereou Street from 6 February 1990

- object (Riga type Fereou Street)
- object (Vasilis type Resident)
- Relationship (livesIn Vasilis, Riga Fereou, t1)

In this example it is clear that Vasilis lived elsewhere before 1990 and that the relationship (livesIn Riga Fereou) is true for a specific time period that starts in 1990. To solve this problem we need the so-called fluents. Fluents call relationships that are true for a certain period of time and only then used to transfer the representation of a relationship from synchronic to diachronic. This is done by changing a relationship from binary to ternary, placing third argument for the period for which the relation is true. So in the example relation (livesIn Riga Fereou) will (livesIn Vasilis, Riga Fereou, t1) where t1 is the period beginning with the date 02/01/1990. The fluents are not supported by languages such as OWL.

1.4 XML

In order to signify the importance of the Web, the influence of metadata was substantial. Metadata is essentially data-about-data whose purpose is to assist us to understand, use and operate

data. In essence serve human-user to more easily clarify the information and its place among a large amount of data. The main language used for describing data on the web is the XML (eXtensible Markup Language).

The XML gives us the ability to create unlimited texts with complex structure and syntax. This makes the information easier to process by computers. Similarly, HTML and XML as markup languages that are using tags. The contents along with each tag called elements. But unlike with HTML, XML is not limited by predefined structures and can describe arbitrary structures and data. The only limitation is the one set of rules for designing text formats that facilitates the design documents.

The main problem with XML documents is due to the arbitrary structure of tags. Assume that you want two websites to exchange or compare data similar to each other, to arise if the structure being-completely-arbitrary likely to be different between the two files (.xml). The problem is called to solve a kind of grammar for defining constraints, called Document Type Definitions (DTD) and the DTD comes to replace the newest standard XML Schema.

The XML Schema is a standard for writing predefined dictionaries and grammars for XML documents. It supposed to work as a structural markup language in XML, increasing the ease of reading an XML file and achieving reusability important files.

1.5 RDF

While XML is the markup language that is widely used in Web and Web2.0, the Semantic Web (Web3.0) that is based on knowledge and not on information, needed a new markup language. This gap fills with the RDF language (Resource Description Framework). This standard was adopted by the W3C for describing information resources and knowledge representation in the online environment. The RDF is based on the idea of identifying objects by using URIs (Uniform Resource Identifiers). The RDF uses a graph model to represent proposals using nodes and arrows.

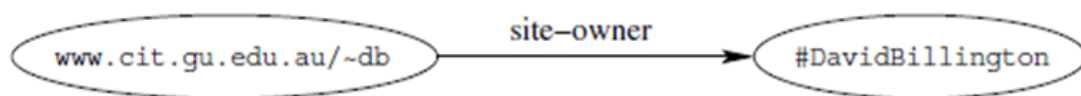
So one sentence as

"David Billington is the owner of the Web page <http://www.cit.gu.edu.au/~db>"

Consisting of:

- a node on the subject
- a node for the object
- and an arrow on the predicate of the subject to the object

To graph model is the following:



The RDF model is capable of expressing virtually any form of knowledge representation in the above manner. If you do not want to recreate the proposal writing can do with triplet (triplet).

```
<http://www.cit.gu.edu.au/~db>  
<http://www.mydomain.org/site-owner>  
<#DavidBillington>
```

1.6 RDFS

RDF Schema (Resource Description Framework Schema, variously abbreviated as RDFS, RDF(S), RDF-S, or RDF/S) is a set of classes with certain properties using

Design and Implementation of a Time Reasoner for Knowledge Representation on RDFS

the RDF extensible knowledge representation language, providing basic elements for the description of ontologies, otherwise called RDF vocabularies, intended to structure RDF resources. These resources can be saved in a triple store to reach them with the query language SPARQL.

The first version was published by the World-Wide Web Consortium (W3C) in April 1998, and the final W3C recommendation was released in February 2004. Many RDFS components are included in the more expressive Web Ontology Language (OWL). RDFS constructs are the RDFS classes, associated properties and utility properties built on the limited vocabulary of RDF.

RDFS Classes

- **rdfs:Resource** is the class of everything. All things described by RDF are resources.
- **rdfs:Class** declares a resource as a class for other resources.

A typical example of an `rdfs:Class` is `foaf:Person` in the Friend of a Friend (FOAF) vocabulary. An instance of `foaf:Person` is a resource that is linked to the class `foaf:Person` using the `rdf:type` property, such as in the following formal expression of the natural language sentence : 'John is a Person'.

ex:John rdf:type foaf:Person

The definition of `rdfs:Class` is recursive: `rdfs:Class` is the `rdfs:Class` of any `rdfs:Class`.

The other classes described by the RDF and RDFS specifications are:

- **rdfs:Literal** – literal values such as strings and integers. Property values such as textual strings are examples of RDF literals. Literals may be plain or typed.
- **rdfs:Datatype** – the class of datatypes. `rdfs:Datatype` is both an instance of and a subclass of `rdfs:Class`. Each instance of `rdfs:Datatype` is a subclass of `rdfs:Literal`.
- **rdf:XMLLiteral** – the class of XML literal values. `rdf:XMLLiteral` is an instance of `rdfs:Datatype` (and thus a subclass of `rdfs:Literal`).
- **rdf:Property** – the class of properties.

RDFS Properties

Properties are instances of the class `rdf:Property` and describe a relation between subject resources and object resources. When used as such a property is a predicate.

Design and Implementation of a Time Reasoner for Knowledge Representation on RDFS

- **rdfs:domain** of an rdf:predicate declares the class of the *subject* in a triple whose second component is the *predicate*.
- **rdfs:range** of an rdf:predicate declares the class or datatype of the *object* in a triple whose second component is the predicate.

For example, the following declarations are used to express that the property `ex:employer` relates a subject, which is of type `foaf:Person`, to an object, which is of type `foaf:Organization`:

ex:employer rdfs:domain foaf:Person

ex:employer rdfs:range foaf:Organization

Given the previous two declarations, the following triple requires that `ex:John` is necessarily a `foaf:Person`, and `ex:CompanyX` is necessarily a `foaf:Organization`:

ex:John ex:employer ex:CompanyX

- **rdf:type** is a property used to state that a resource is an instance of a class. A commonly accepted qname for this property is "a".
- **rdfs:subClassOf** allows to declare hierarchies of classes.

For example, the following declares that 'Every Person is an Agent':

foaf:Person rdfs:subClassOf foaf:Agent

Hierarchies of classes support inheritance of a property domain and range (see definitions in next section) from a class to its subclasses.

- **rdfs:subPropertyOf** is an instance of `rdf:Property` that is used to state that all resources related by one property are also related by another.
- **rdfs:label** is an instance of `rdf:Property` that may be used to provide a human-readable version of a resource's name.
- **rdfs:comment** is an instance of `rdf:Property` that may be used to provide a human-readable description of a resource.

Utility properties

- **rdfs:seeAlso** is an instance of `rdf:Property` that is used to indicate a resource that might provide additional information about the subject resource.
- **rdfs:isDefinedBy** is an instance of `rdf:Property` that is used to indicate a resource defining the subject resource. This property may be used to indicate an RDF vocabulary in which a resource is described.

RDFS entailment

An entailment regime defines by RDFs (OWL, etc.) not only which entailment relation is used, but also which queries and graphs are well-formed for the regime. The RDFS entailment is a standard entailment relations in the semantic web.

Design and Implementation of a Time Reasoner for Knowledge Representation on RDFS

For example, the following declares that 'Dog1 is an animal', 'Cat1 is a cat', 'Zoos host animals' and 'Zoo1 hosts the 'Cat2':

```
ex:dog1      rdf:type      ex:animal
ex:cat1      rdf:type      ex:cat
zoo:host     rdfs:range    ex:animal
ex:zoo1     zoo:host      ex:cat2
```

But this graph is not well formed because the system cannot guess that a cat is an animal. We have to add 'Cats are animals' to do a well-formed graph with:

```
ex:cat      rdfs:subClassOf  ex:animal
```

The correct example:

In English	The graph
<ul style="list-style-type: none"> • Dog1 is an animal • Cat1 is a cat • Cats are animals • Zoos host animals • Zoo1 hosts the Cat2 	
RDF/turtle	
<pre>@PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> . @PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> . @PREFIX ex: <http://example.org/> . @PREFIX zoo: <http://example.org/zoo/> . ex:dog1 rdf:type ex:animal . ex:cat1 rdf:type ex:cat . ex:cat rdfs:subClassOf ex:animal . zoo:host rdfs:range ex:animal . ex:zoo1 zoo:host ex:cat2 .</pre>	

If your triple store (or RDF database) implements the regime entailment of RDF and RDFS, the SPARQL query as follows (the keyword "a" is equivalent to rdf:type in SPARQL):

```
PREFIX ex: <http://example.org/>
SELECT ?animal
WHERE
{ ?animal a ex:animal . }
```

Gives the following result with *cat1* in it because the Cat's type inherits of Animal's type:

animal
<http://example.org/dog1>
<http://example.org/cat1>
<http://example.org/cat2>

1.7 Protégé

Protégé is a tool made from Stanford University for the purpose of creating and maintaining ontologies. It is open source, free and fully customizable. Today many companies, government organizations and colleges use Protégé to create ontologies, using OWL. Protégé is capable of exporting the ontologies in RDF(S), OWL, N3, Turtle and RDF(S)/XML syntax. Protégé also supports plug-ins to expand its capabilities.

The user interface looks simple but it is very powerful. It consists of Tabs mainly, that represent the Individuals, Entities, Classes, Object properties etc. of the ontology. Another Tab is for writing SparQL queries and another for representing the Graph Model of the ontology.

2 Artificial Intelligence



2.1 Artificial Intelligence and Logic

AI (Artificial Intelligence) is the area of computer science that deals with the design of intelligent systems, namely computing systems that exhibit characteristics with human behavior. The goal of Artificial Intelligence is to solve problems of computing and the equation that describes it is “AI = Knowledge Representation + Search”.

Knowledge Representation is a way of representing knowledge about a problem, with the aim of describing the problem and automating the reasoning to solve it. The search terms of various algorithms that automate the process of solving a problem, seeking solutions using the appropriate representation of knowledge that describes the problem. The search algorithms, together with knowledge representation, form the core of every application of AI. The First Order Predicate Calculus is one of the most popular methods of knowledge representation.

2.2 Knowledge Representation

The Knowledge Representation is the area of Artificial Intelligence that deals with how can knowledge be represented better and more efficiently. It is a field that attracts great interest from the famous search engines (Google, Bing, Yahoo) and services like Wolfram Alpha and personal assistants of mobile phones (Apple Siri, Google Voice, Microsoft Cortana) targeting the most accurate performance information required than to return a large volume of information. To do this, however, the search engines should look for information based on the meaning of propositions questions instead of keywords.

Every computer system that exhibits intelligent behavior involves two basic components. The first is a knowledge base and the second is a mechanism of inference. Knowledge is not programmed into the system, but explicitly described in the knowledge base (KB) with the help of a standard-strict language, called Knowledge Representation Language (KRL). The knowledge base consists of a set of KRL expression proposals that describe the knowledge embedded in the system.

A knowledge representation language can be defined as a set of syntactic and semantic conventions enables the description of some knowledge. This set is accompanied by another set of rules that allows the efficient handling of this knowledge. A KRL has two key elements. One is the syntax or notation and determine how to form the correct expressions of the language. The syntax of a language includes: a) a set of primary symbols (vocabulary) and b) a set of grammatical rules (syntax rules) to form expressions of the language. Expressions of KRL with correct syntax called well-formed

expressions or well-formed formulas (WFF). The second aspect is the semantics or significant of a KRL, which determines whether an expression is true or false. It consists of a set of rules (semantic rules) fourteen (14) of which define the concept of a complex of KRL concepts of individual elementary expressions which they constitute.

The KRL handles the WFF language to produce new knowledge, in the sense that knowledge not explicitly described, but inherent in the KB, becomes evident. E.g. of the proposals "every man is mortal", "Socrates is a man" may be inferred "Socrates is mortal", which however is not entirely new knowledge, but revelation knowledge implied in the preceding two sentences. This manipulation - knowledge processing is done using certain abstract rules, called rules of inference rules or conclusive and general i.e. not dependent on specific knowledge located in the knowledge base. .

The most popular methods of knowledge representation fall into three main categories: Logic, structured knowledge representations and rules (if-then rules). In Logic belongs the propositional logic, the predicate logic and the disjunctive form of logical (clausal form of logic). On Structured Knowledge Representation belong semantic networks, frames, the conceptual dependency and scripts.

2.3 Logic

Logic provides a way to clarify and standardize the process of human thought and offers an important and convenient method for representing and solving problems. The need to use a strictly specific language, with a mathematical concept, originated by the inadequacy to use natural language in computer systems. Instead, the logic provides a clear, accurate and simple to language syntax, and the possibility of generating new knowledge from existing.

Logic is defined as the study of correct inference. A minimum requirement for proper inference is to maintain the truth, i.e. the requirement of true cases - recommendations exported to true conclusions - recommendations. This is why Logic is defined as a minimum as the study of the preservation of truth in drawing conclusions (inference). To define a logical language we need to define three key elements: i) the structure, ii) the significance, and iii) the probative theory (conclusive rules).

There are two main types of formal logic, the Propositional Calculus and Predicate Calculus. The Propositional Calculus or propositional logic uses full sentences as building blocks, while the Predicate Calculus and Predicate Logic analyzes a proposal to more structural units. Most advanced and most useful kind of logic, especially for Artificial Intelligence applications, is the Predicate Logic and more specifically the First Order Predicate Calculus, abbreviated FOL.

2.4 Propositional Logic

The propositional logic is the simplest kind of logic. In this logic every event of the real world is represented by a logical sentence, which is characterized as either true or as false. Reasonable proposals usually represented by characters: P, Q, R, etc. and called interest-free (atoms). The interest-free can be combined using logical symbols or connectives, which are shown in the table below along with the names and their explanations:

Symbol	Name/ Meaning
\wedge	Conjunction (logical AND)
\vee	Disjunction(logical OR)
\neg	Negation (logical NOT)
\rightarrow	Material implication (If - Then)
\leftrightarrow	Biconditional (If Only)

The resulting complex sentences correctly called structured types. The logical value of properly structured types calculated using truth tables or proof. Examples of knowledge representation using propositional logic shown below. In each proposal (called event) we want to represent a corresponding Latin character:

P: «Nick is a developer"

Q: «Nick has a computer"

The representation of the knowledge that if Nick is a developer, and has since computer is made by combining the above two proposals through the appropriate binder in $P \rightarrow Q$: If "Nick is a developer" then "Nick has a computer." Assuming that the proposals P and Q are true, then the correctly structured type $P \rightarrow Q$ is true. In the two following proposals represented knowledge concerning the properties of a given triangle ABC:

R: <<The triangle ABC is equilateral>>

V: <<The triangle ABC has all sides of equal>>

The equivalence of the R and V is indicated by the following well-structured formula: $R \leftrightarrow V$: "The triangle ABC is equilateral" if and only if "The triangle ABC has all sides of equal".

The advantages of using propositional logic for knowledge representation is the simplicity of preparation and the fact that it can always come to a conclusion. But an essential drawback is the lack of generality leads to voluminous knowledge representations, and each event must be represented by a separate logical proposal.

2.5 Predicate Logic (First Order Logic)

Predicate Logic or First Order Logic solves the data accessibility problem of the events of Propositional Logic. For example in predicate logic the proposal <Nick is a programmer> is represented as programmer (Nick). This representation allows the object data to be inferred for the extraction of new knowledge.

Predicate logic expands Propositional logic, importing terms, predicates and quantifiers. A fact is represented with a personal type of the P (1, 2.....A, A, An) form, where P is the predicate and the rest are the arguments. Every argument can me a constant, variable or functional term. Functional terms have the f(1,2,..., nt, t) form, where f is the functional symbol and the rest are the arguments.

Predicate logic connectives are the same with the ones in Propositional logic except two more symbols called quantifiers.

Symbol	Name/ Meaning
\forall	Universal Quantifier ($\forall x$ means : for every x)
\exists	Existential Quantifier ($\exists x$ means : there is x)

The quantifiers are increasing the expressiveness of predicate logic. New kinds of sentences can be created such as:

<Every human has a name> as $(\forall x)(\exists y \text{ human}(x) \rightarrow \text{name}(x))$.

<Every basketball player is tall> as $(\forall x (\text{basketball_player}(x) \rightarrow \text{tall}(x)))$.

The advantages of predicate logic are summarized in the correspondence with the natural language, the efficient expression quantification of concepts with appropriate quantifiers and its ability to capture the generality. One major drawback of logic is generally the inability to express the uncertainty, as each sentence can be true or false without being given the chance to express fuzzy values. Disadvantages include also the additivity of effects, i.e. a drawn conclusion without added knowledge to enable revision if you later found to be incorrect (monotonic logic).

To determine the various elements of predicate logic we need to clarify the meaning of the following domain. Domain D (Universe of Discourse), is called the set of objects-entities associated with the knowledge that we want to represent. Below are the basic syntax rules of predicate logic.

- A set of constants (constants): $\{c_i\}$, element of D. A constant represents a specific object in D.
- The logical constants true and false: $\{T, F\}$.
- A set of variables (variables): $\{v_i\}$, subset of D. A variable represents an object in D without naming what.
- A set of functions (functions): $\{f_i\}: n \text{ DD} \rightarrow$. A function n variable is a bi-unambiguous display corresponding set of entities in an entity. The function that refers to an entity associated with variables - entities that are its terms.

- A set of predicates : $\{P_i\}: n D \rightarrow \{T, F\}$. A predicate n arguments or positions is a mapping of a sequence of n objects in the domain $D \{T, F\}$, expresses a correlation between objects. If n objects associated with each other's way that indicates the predicate, then it takes the value of T , otherwise takes value F .

- The logic interfaced (connectives): not (not \neg), or (or \vee), and (and \wedge), implies (implies \Rightarrow) and equivalent (equivalent \Leftrightarrow). We saw before in Predicate Logic with a small difference in the last two symbols are, however, equivalent to the previous ones.

- Two quantifiers: the catholic (universal \forall) and existential (existential \exists). And quantifiers we saw in the section of predicate logic and the table above.

Constants, logical constants, variables, functions and predicates make up the vocabulary of predicate logic. As we saw before, the basic building block of a logical expression in predicate logic is the individual expression or person, and has the form $P(1, \dots, nt)$, where P a predicate arguments n and $1. \dots, Nt$ the conditions.

A term is recursively defined as follows:

- i. A constant is a term.
- ii. A variable is a term.
- iii. If f is a function and n variables $1. \dots, Nt$ are terms, then $f(1, \dots, nt)$ is a term.
- iv. All conditions produced by applying the rules (i), (ii), (iii).

Based on the foregoing, we now define a strictly well-formed expression (WFE) or simply an expression in predicate logic:

- i. A person is a WFE.
- ii. If F and G are WFE, then $\neg \vee \wedge \Rightarrow \Leftrightarrow$ FFGFGFGFG, $()$, $()$, $()$ and $()$ is WFE.
- iii. If F is a WFE and a free variable x in F , then $() \forall x F$ and $() \exists x F$ is WFE.
- iv. The WFE created only a finite number of applications of (i), (ii) and (iii).

For example, if *the greater* is a predicate that expresses the relationship 'bigger', then the greater (3, 2) is a person who is true (T), while the greater (1, 3) is false (F). If x, y are variables then the expression $() () \text{ greater } (,) \forall \exists x y x y$ is a WFE and means "for every x there is y such that the greater (x, y) to be true", in other words "for every x there is y such that x is greater than y ».

Each quantifier has a scope, which is the expression to which it applies. For example, the last expression of the range of the x and y is the expression greater (x, y). A variable is called bound in an expression, if and only if an instance of this expression is within the scope of a quantifier that identifies it. A variable not bound is called free. For example, in the expression $() (,) \forall x P x y$, the x variable is bound, while y is free. A proposal containing free variables called open proposal while a sentence containing no free variables called closed proposal.

Design and Implementation of a Time Reasoner for Knowledge Representation on RDFS

Here is an example which is represented in predicate logic knowledge about the characteristics of different species. This knowledge is reflected in the following set of sentences:

- i. Any animal that has fur or produce milk is a mammal.
- ii. Each animal that has wings and lays eggs is a bird.
- iii. Every mammal that eats meat or has sharp teeth are carnivorous.
- iv. Every carnivore with brown-orange tiger has stripes are.
- v. Every carnivore with orange-brown color that has black dots are cheetahs.
- vi. Every bird that does not fly and swims is penguin.

Below are representations of the above proposals in predicate logic:

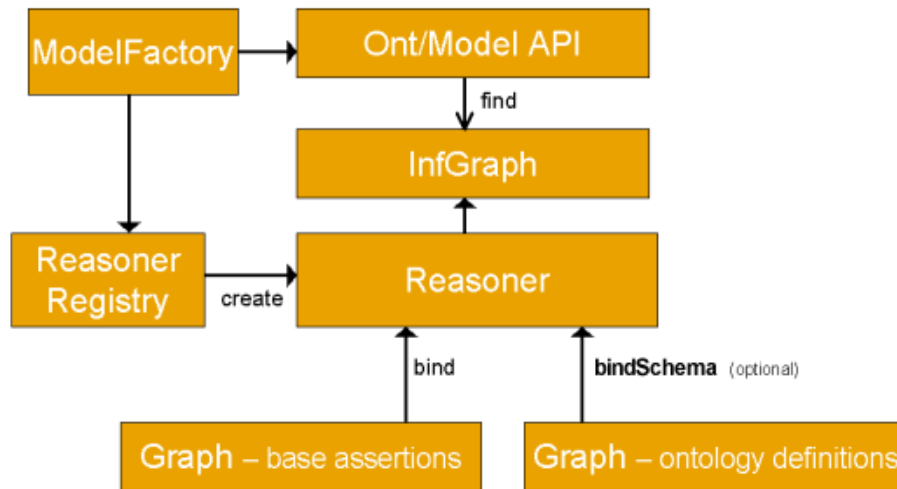
- i. $(\forall x) (\text{has}(x, \text{fur}) \vee \text{produces}(x, \text{milk})) \rightarrow \text{mammal}(x)$
- ii. $(\forall x) (\text{has}(x, \text{wings}) \wedge \text{lays}(x, \text{eggs})) \rightarrow \text{bird}(x)$
- iii. $(\forall x) (\text{mammal}(x) \wedge (\text{eats}(x, \text{meat}) \vee \text{has}(x, \text{sharp_teeth}))) \rightarrow \text{carnivore}(x)$
- iv. $(\forall x) (\text{carnivore}(x) \wedge \text{color}(\text{brown_orange}, x) \wedge \text{has}(x, \text{black_dots})) \rightarrow \text{tiger}(x)$
- v. $(\forall x) (\text{carnivore}(x) \wedge \text{color}(\text{brown_orange}, x) \wedge \text{has}(x, \text{black_dots})) \rightarrow \text{Cheetah}(x)$
- vi. $(\forall x) (\text{bird}(x) \wedge \neg \text{flies}(x) \wedge \text{swims}(x)) \rightarrow \text{penguin}(x)$

The main advantages of predicate logic and logical languages in general are

- they have clear significance,
- they have great expressiveness,
- And they provide declarative representation.

The concepts of logical propositions can be specified, which enables control of the correct representation of knowledge. Disadvantages, on the other hand, is inefficiency, indecisiveness, the inability for representation of procedural knowledge and monotonicity.

3 Fluents and Reasoners



3.1 Reasoner

Reasoner or rule engine is a software that extracts logical conclusions from a set of rules written in the form of axioms. The basic services that make reasoners popular when developing Semantic Web applications is the consistency checking of the knowledge base, the instance checking that the calculation of the classes to which it belongs in every instance of our shape and categorizing classes or classification. The most popular reasoners are divided into three categories:

- Commercial software such as Bossam, a rules engine that supports OWL Ontologies and SWRL rules and RuleML.
- Free closed source software including machines like Cyc, Kon2 and the IBL (Internet Business Logic) software,
- and free open source software such as the reasoner Cwm, the rules engines Prova, Flora-2, Drools and the Jena Framework which I will refer extensively later.

3.2 Situation Logic

The situation calculus is a logic formalism designed for representing and reasoning about dynamical domains. It was first introduced by John McCarthy in 1963. The situation calculus represents changing scenarios as a set of first-order logic formulae.

The basic elements of the calculus are:

- The actions that can be performed in the world
- The fluents that describe the state of the world
- The situations

A domain is formalized by a number of expressions like:

- The action precondition axioms, one for each action
- Successor state axioms, one for each fluent
- Axioms describing the world in various situations
- And the foundational axioms of the situation calculus

3.3 Fluents

In artificial intelligence the fluent is a condition that changes over time. The fluents can be represented as first-order logic. For example, the condition "the box is on the table" can be represented as $\text{On}(\text{box}, \text{table})$ in an ontology, but if you want to add the time factor should write $\text{On}(\text{box}, \text{table}, t)$ where t is time. The representation of fluents is used in Situation Calculus series, replacing old with new situations. A fluent can be represented by a function without the time variable. In Example $\text{On}(\text{box}, \text{table})$, On may be a function instead of a predicate. The predicates conversion to functions in first-order logic called reification.

Statements whose truth value may change are modeled by relational fluents, predicates which take a situation as their final argument. Also possible are functional fluents, functions which take a situation as their final argument and return a situation-dependent value. Fluents may be thought of as properties of the world.

3.4 Allen's Integral Algebra

In 1983 James F. Allen published a paper in which he proposed thirteen basic relations between time intervals that are distinct, exhaustive, and qualitative.

- **distinct** because no pair of definite intervals can be related by more than one of the relationships
- **exhaustive** because any pair of definite intervals are described by one of the relations
- **qualitative** (rather than quantitative) because no numeric time spans are considered

These relations and the operations on them form *Allen's interval algebra*.

Thirteen basic relations

Allen's thirteen basic relations are illustrated in Table 1. This table shows all the possible relations that two definite intervals can have. Each one is defined graphically by a diagram relating two definite intervals *a* and *b*, with time running → from left to right. For example, the first diagram shows that "*a* precedes *b*" means that *a* ends before *b* begins, with a gap separating them; the second shows that "*a* meets *b*" means that *b* ends when *a* begins.







precedes	meets	overlaps	finished by	contains	starts	equals	started by	during	finishes	overlapped by	met by	preceded by
p	m	o	F	D	s	e	S	d	f	O	M	P

The basic relations are listed in Table 1 sorted by the degree to which *a* begins before *b* and then within that by the degree to which *a* ends before *b*. We will commonly list them in this order (pmoFDseSdfOMP), as it makes the relations easier to remember and simplifies comparison of general relations.

Six pairs of the relations are converses. For example, the converse of "*a* precedes *b*" is "*b* preceded by *a*"; whenever the first relation is true, its converse is true also. Table 2 lists the relations with each one beside its converse. The thirteenth, "equals", is its own converse. Each pair of converse relation symbols consists of the lowercase and uppercase of the same letter (e.g. p and P; the uppercase letters represent the relations Allen defined as converses).

Relation	Converse	
precedes	(p)	(P) preceded by
meets	(m)	(M) met by
overlaps	(o)	(O) overlapped by
finished by	(F)	(f) finishes
contains	(D)	(d) during
starts	(s)	(S) started by
equals (e)		

Example:

a	(pmMP)	b	
"John was in the room"	p		"I touched the light switch"
	m		
	M		
	P		
b	(mo)	c	
"I touched the light switch"	m		"The light was on"
	o		

The basic relations describe relations between definite intervals. Indefinite intervals whose exact relation may be uncertain are described by a set of all the basic relations that may apply. We call such a set of basic relations a general Allen relation, or just an Allen relation.

For example, "John was not in the room when I touched the switch to turn on the light" .

Let

- a be the time John was in the room,
- b be the time I touched the light switch, and
- c be the time the light was on.

Then we can say a (pmMP) b , that is, a precedes, meets, is met by, or is preceded by b ; and b (mo) c , that is, b meets or overlaps c . Table 3 shows these relations.

There is a general relation for every combination of the thirteen basic relations: 2^{13} or 8192 of them. Each of the basic relations is a relation, of course, as are all their combinations. The full relation (pmoFDseSdfOMP) holds between two intervals about whom nothing is known. The empty relation () has no meaning in terms of relations between actual intervals, but is the result of some operations on interval relations and is needed for sub-algebras of Allen's interval algebra (discussed below).

Complement

Complement examples
$\sim(p) = (moFDseSdfOMP)$
$\sim(pmoFD) = (seSdfOMP)$
$\sim() = (pmoFDseSdfOMP)$

The complement $\sim r$ of a relation r is the relation consisting of all basic relations not in r . From the definition of complement, we see that the converse operation is its own inverse; for every relation r ,
 $\sim(\sim r) = r$

Composition

Composition examples
$(m).(m) = (p)$
$(pm).(pm) = (p)$
$(oFD).(oFDseS) = (pmoFD)$

The composition $(r.s)$ of two relations (r) and (s) is the relation that holds between a and c if there is a b such that $a(r)b$ and $b(s)c$; we then write $a(r.s)c$.

Calculation of composition is not simple like the other operations in this section. It can be determined by going back to the definitions of the relations, and working from there; or by determining the composition of each basic relation from r with each basic relation from s (using a [table](#), perhaps), and taking the union of the results; or by using the "allen" command.

Composition is not commutative but is both left and right associative, and distributes over union (as seen in the procedure for calculating composition using a table of composition of basic relations).

Converse

Converse examples
$!(p) = (P)$
$!(pmoFD) = (dfOMP)$
$!(mM) = (mM)$
$!() = ()$

The converse $!r$ of a relation r is the relation consisting of the converses of all basic relations in r . From the definition of converse, we see that the converse operation is its own inverse; for every relation r ,

$$!(!r) = r$$

Intersection

Intersection examples
$(pmo) \wedge (FDseS) = ()$
$(pFsSf) \wedge (pmoFD) = (pF)$
$(pmo) \wedge (pmo) = (pmo)$

The intersection ($r \wedge s$) of two relations (r) and (s) is the set-theoretic intersection of the two relations; it is the relation composed of all basic relations that are in both (r) and (s). Intersection is commutative and associative.

Union

Union examples
$(pmo) + (FDseS) = (pmoFDseS)$
$(pFsSf) + (pmoFD) = (pmoFDsSf)$
$(pmo) + (pmo) = (pmo)$

The intersection ($r + s$) of two relations (r) and (s) is the set-theoretic intersection of the two relations; it is the relation composed of all basic relations that are in either (r) or (s). Union is commutative and associative.

4 Jena

4.1 Jena API

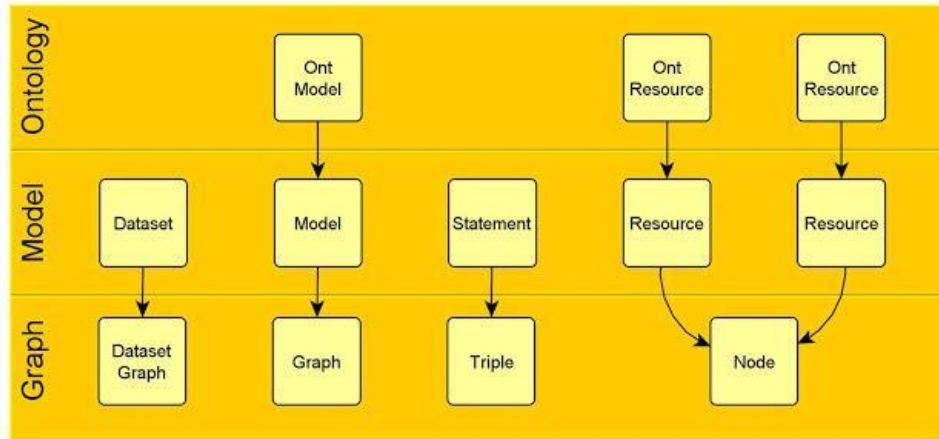


Figure 1 Jena API layers

Jena API is an open source Java Framework used for the creation of Web Services. It can be used to offer storage, inference and query ontologies of RDF/RDFS or OWL syntax. It was first developed by Hewlett Packard from 2000 until 2009. Then it was moved to the Apache Software Foundation until today. The last stable release was published in September of 2013 and is the 2.11.0. The language it uses to query the ontologies is similar to SQL but it is specifically developed for the Semantic web. It is called SparQL and it comes preinstalled in the API. The Jena API supports DAML, +OIL, N3, Turtle and OWL syntax.

The Jena API includes:

- An RDF API
- An OWL API
- RDF, OWL and a generic purpose inference engine
- The SparQL query engine
- Data storage capabilities

The RDF data model expresses the data in graph model transforming the sentences to triplets consisting of Subject-Predicate-Object. There are usually more than one models in an RDF file. Jena is implemented in three levels, the graph layer, the model layer and the ontology layer. The basic layer in Jena is the Graph layer (SPI) and it is where the RDF implementation happens.

Example of a triplet created in the Graph layer:

```
Node s = Node.createURI("Vasilis");
Node p = Node.createURI("isLivingin");
Node o = Node.createURI("Greece");
Triple triple = new Triple(s, p, o);
graph.add(triple);
```

The second layer is the model layer. We will create the same triplet now to see the difference in the implementation and the advance power of the resources.

```
Resource s = model.createResource("Vasilis");
Property p = model.createProperty("isLivingin");
Resource o = model.createResource("Greece");
model.add(subject, predicate, object);
Statement statement = model.createStatement(subject, predicate, object);
```

The last Jena layer, the ontology model (OntModel) is the third from the bottom. In this layer we have the ability to infer results. This means that instead of the standard triplets of our model we can also use the new inferred triplets that we inferred using rules. For example, if $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$ is inferred. The last triplet is inferred by the use of a reasoner.

Jena Basics:

Loading a simple ontology:

```
public OntologyLoader(String fileName) {
// ontology that will be used
String ontologyUrl;
setOntologyUrl("file:/// " + fileName);
// create an empty ontology model (OntModel)
model = ModelFactory.createOntologyModel(OntModelSpec.OWL_DL);
// read the file
model.read(ont);
}
```

Loading an ontology with Pellet Reasoner:

Design and Implementation of a Time Reasoner for Knowledge Representation on RDFS

```
public OntologyLoader(String fileName) {  
    private String uri = "";  
    private String ontologyUrl;  
    private String fileName;  
    private OntModel ontModel;  
    private OntDocumentManager dm;  
    setOntologyUrl("file:/// " + fileName); // the third slash is needed  
for windows xp  
    OntModel base = ModelFactory.createOntologyModel(); //empty model  
    dm = base.getDocumentManager(); // used in this class  
    dm.addAltEntry(uri, ontologyUrl); // used in this class  
    base.read(uri);  
    ontModel =  
ModelFactory.createOntologyModel(PelletReasonerFactory.THE_SPEC, base);  
}
```

Reading Classes from the ontology:

```
public void readClasses() {  
    ExtendedIterator<OntClass> iter = ontoModel.listClasses();  
    while(iter.hasNext()) {  
        OntClass ontClass = iter.next();  
        System.out.println("CLASS : "+ontClass.getLocalName());  
    }  
}
```

Reading the instances of a class from the ontology:

```
OntClass newClass = model.getOntClass( classUrl );  
Iterator instances = newClass.listInstances();
```

Reading the Datatype & Object Properties from the ontology:

```
public void readProperties() {  
    ExtendedIterator<DatatypeProperty> iter =  
ontoModel.listDatatypeProperties();  
    ExtendedIterator<ObjectProperty> iter2 =  
ontoModel.listObjectProperties();  
    while(iter.hasNext()) {  
        DatatypeProperty dataProperty = iter.next();  
        System.out.println( dataProperty.getLocalName() );  
    }  
    while(iter.hasNext()) {  
        ObjectProperty objProperty = iter.next();  
        System.out.println( objectProperty.getLocalName() );  
    }  
}
```

Design and Implementation of a Time Reasoner for Knowledge Representation on RDFS

Reading Statements from the ontology:

```
Public void readAllStatements(OntModel model){
    StmtIterator iter;
    Statement stmt;
    iter = model.listStatements();
    while (iter.hasNext()) {
        stmt = iter.next();
// using the statement to read SUBJECT-PREDICATE-OBJECT
        Property predicate;
        Resource subject;
        RDFNode obj;
        subject = stmt.getSubject(); System.out.println("Subject = " +
        subject.getURI());
        predicate = stmt.getPredicate(); System.out.println("Predicate =
        "+predicate.getLocalName());
        obj = stmt.getObject(); System.out.println("Object = " +
        obj.toString());
    }
}
```

Add new statements to the ontology:

```
. . .
Model model;
String namespace = "http://www.example.org"; . . .
Resource res =
model.createResource("http://www.example.com/companies#Company1")
Property property1 = model.createProperty(namespace,
"numOfEmployees");
res.addProperty(property1 , 25);
Property property2 = model.createProperty(namespace, "Location")
res.addProperty( property2 , "Athens");
```

Examples of the Pellet Reasoner usage:

```
// Creating the model using Pellet
ontModel =
ModelFactory.createOntologyModel(PelletReasonerFactory.THE_SPEC,base);
// Creating a new Pellet reasoner
PelletInfGraph reasoner;
reasoner = (PelletInfGraph) ontModel.getGraph();
// Calling the reasoner whenever needed
Reasoner.classify();
Reasoner.realize();
```

Executing a Query examples:

```
// Create a new query

String queryString = "PREFIX ex1: <http://example.org/ex1/> " + "SELECT
?x " +
    "WHERE {" + " ?x ex1:employeeName \"John\" }";
Query query = QueryFactory.create(queryString);

// Execute the query and obtain results. model is an OntModel.

QueryExecution qe = QueryExecutionFactory.create(query, model);
ResultSet results = qe.execSelect();

// Output query results

ResultSetFormatter.out(System.out, results, query);

// Important - free up resources used running the query

qe.close();
```

4.2 Inference Engines

The Jena API and more specifically the inference engine is designed to support reasoners. Those reasoners are used to extract New Knowledge in the form of RDF triplets. The inference engine supports RDFS and OWL syntax. The API also includes a general purpose rule engine for custom rule sets written in the RuleML syntax.

In the Jena API are includes the following reasoners:

1. Transitive reasoner
2. RDFS rule reasoner
3. OWL, OWL Mini, OWL Micro Reasoners
4. Generic rule reasoner

The generic rule reasoner includes the RDFS and Owl reasoners but it can also use custom rules from an outside file. It supports forward chaining, backward chaining and hybrid rules. The generic rule reasoner connects with a data model to be used for querying the ontologies.

Every rule is defined as a Java rule object and includes premises, conclusions optional name and an optional direction. Every term (Clause Entry) is either a triple pattern, an extended triple pattern or a call to built-in primitive.

Another common thing in a rule file is the prefixes, usually on top. Those are used locally to replace URIs with a local variable for greater readability and easier editing. It is important here to state that @prefix is different from @include. The second is a command that includes another rule file like: (RDFS, OWL, OWLMini, OWLMacro etc.).

A stack of rules is called a rule set. The rule files are loaded in the program with the following command:

```
List rules = Rule.rulesFromURL ("file: myfile.rules");
```

Or

```
BufferedReader br = / open reader /;  
List rules = Rule.parseRules( Rule.rulesParserFromReader(br) );
```

Or

```
String ruleSrc = / list of rules in line /  
List rules = Rule.parseRules( ruleSrc );
```

Forward chaining engine

```
[DescriptionOrNameOfRule:  
  
  (condition to be met)  
  
  (another condition)  
  
->  
  
  (fact to assert)  
  
  (another fact to assert)  
  
]
```

If the reasoner is configured to run in forward mode then only the forward chaining engine will be used. The first time the inference Model is queried (or when an explicit `prepare()` call is made) then all of the relevant data in the model will be submitted to the rule engine. Any rules which fire that create additional triples do so in an internal deductions graph and can in turn trigger additional rules. There is a remove primitive that can be used to remove triples and such removals can also trigger rules to fire in removal mode. This cascade of rule firings continues until no more rules can fire. It is perfectly possible, though not a good idea, to write rules that will loop infinitely at this point.

Backward chaining engine

```
[DescriptionOrNameOfRule:  
  
  (fact to assert)  
  
  (another fact to assert)  
  
<-  
  
  (condition to be met)  
  
  (another condition)  
  
]
```

If the rule reasoner is run in backward chaining mode it uses a logic programming (LP) engine with a similar execution strategy to Prolog engines. When the inference Model is queried then the query is translated into a goal and the engine attempts to satisfy that goal by matching to any stored triples and by goal resolution against the backward chaining rules.

GenericRuleReasoner configuration

As with the other reasoners there are a set of parameters, identified by RDF properties, to control behavior of the `GenericRuleReasoner`. These parameters can be set using the `Reasoner.setParameter` call or passed into the Reasoner factory in an RDF Model.

The primary parameter required to instantiate a useful `GenericRuleReasoner` is a rule set which can be passed into the constructor, for example:

```
String ruleSrc = "[rule1: (?a eg:p ?b) (?b eg:p ?c) -> (?a eg:p ?c)]";
List rules = Rule.parseRules(ruleSrc);
...
Reasoner reasoner = new GenericRuleReasoner(rules);
```

A short cut, useful when the rules are defined in local text files using the syntax described earlier, is the `ruleSet` parameter which gives a file name which should be loadable from either the classpath or relative to the current working directory.

Basic Rule syntax:

```

Rule      :=  bare-rule .
           or  [ bare-rule ]
           or  [ ruleName : bare-rule ]

bare-rule :=  term, ... term -> hterm, ... hterm    // forward rule
           or  bhterm <- term, ... term             // backward rule

hterm     :=  term
           or  [ bare-rule ]

term      :=  (node, node, node)                    // triple pattern
           or  (node, node, functor)                // extended triple pattern
           or  builtin(node, ... node)              // invoke procedural primitive

bhterm    :=  (node, node, node)                    // triple pattern

functor   :=  functorName(node, ... node) // structured literal

node      :=  uri-ref                               // e.g. http://foo.com/eg
           or  prefix:localname                     // e.g. rdf:type
           or  <uri-ref>                            // e.g. <myscheme:myuri>
           or  ?varname                             // variable
           or  'a literal'                          // a plain string literal
           or  'lex'^^typeURI                       // a typed literal, xsd:* type names supported
           or  number                                // e.g. 42 or 25.5
    
```

Summary of parameters

Parameter	Values	Description
PROPruleMode	"forward", "forwardRETE", "backward", "hybrid"	Sets the rule direction mode as discussed above. Default is "hybrid".
PROPruleSet	filename-string	The name of a rule text file which can be found on the classpath or from the current directory.
PROPenableTGCCaching	Boolean	If true, causes an instance of the TransitiveReasoner to be inserted in the forward dataflow to cache the transitive closure of the subProperty and subClass lattices.
PROPenableFunctorFiltering	Boolean	If set to true, this causes the structured literals (functors) generated by rules to be filtered out of any final queries. This allows them to be used for storing intermediate results hidden from the view of the InfModel's clients.
PROPenableOWLTranslation	Boolean	If set to true this causes a procedural preprocessing step to be inserted in the dataflow which supports the OWL reasoner (it translates intersectionOf clauses into groups of backward rules in a way that is clumsy to express in pure rule form).
PROPtraceOn	Boolean	If true, switches on exhaustive tracing of rule executions to the <i>log4j info</i> appender.
PROPderivationLogging	Boolean	If true, causes derivation routes to be recorded internally so that future getDerivation calls can return useful information.

Builtin primitives

The procedural primitives which can be called by the rules are each implemented by a Java object stored in a registry. Additional primitives can be created and registered - see below for more details. Each primitive can optionally be used in either the rule body, the rule head or both. If used in the rule body then as well as binding variables (and any procedural side-effects like printing) the primitive can act as a test - if it returns false the rule will not match. Primitives using in the rule head are only used for their side effects.

The set of built-in primitives available at the time writing are:

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), greaterThan(?x, ?y) le(?x, ?y), ge(?x, ?y)	Test if x is <, >, <= or >= y. Only passes if both x and y are 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

	<p>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</p> <pre>regex('foo bar', '(.*) (.*)', ?m1, ?m2)</pre> <p>will bind m1 to "foo" and m2 to "bar".</p>
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 behaviour 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, can not 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 which are only used for inference control and do not "pollute" the inference results.

4.3 SparQL

SPARQL (SPARQL Protocol and RDF Query Language) is an RDF query language, that is, a query language for databases, able to retrieve and manipulate data stored in Resource Description Framework format. It was made a standard by the RDF Data Access Working Group (DAWG) of the World Wide Web Consortium, and is recognized as one of the key technologies of the semantic web. On 15 January 2008, SPARQL 1.0 became an official W3C Recommendation and SPARQL 1.1 in March, 2013.

SPARQL allows for a query to consist of triple patterns, conjunctions, disjunctions, and optional patterns. Implementations for multiple programming languages exist. According to Sir Tim Berners-Lee "SPARQL will make a huge difference" making the web machine-readable. There exist tools that allow one to connect and semi-automatically construct a SPARQL query for a SPARQL endpoint, for example ViziQuer. In addition, there exist tools that translate SPARQL queries to other query languages, for example to SQL and to XQuery. SPARQL City's SPARQLverse also allows queries directly against non-SPARQL databases such as MongoDB and Cassandra, representing their data as though it is RDF.

SPARQL allows users to write queries against data that can loosely be called "key-value" data, more specifically it is data that follows the RDF specification of the W3C. The entire database is thus a set of "subject-predicate-object" triples. This is analogous to some NoSQL database's usage of the term "document-key-value", such as MongoDB.

RDF data can also be considered in SQL relational database terms as a table with three columns - the subject column, the predicate column and the object column. Unlike relational databases, the object column is heterogeneous, the per-cell data type is usually implied (or specified in the ontology) by the predicate value. Alternately, again comparing to SQL relational, all of the triples for a given subject could be represented as a row, with the subject being the primary key and each possible predicate being a column and the object is the value in the cell. However, SPARQL/RDF becomes easier and more powerful for columns that could contain multiple values (like "children"), and where the column itself could be a joinable variable in the query, rather than directly specified.

SPARQL thus provides a full set of analytic query operations such as JOIN, SORT, AGGREGATE for data whose schema is intrinsically part of the data rather than requiring a separate schema definition. Schema information (the ontology) is often provided externally, though, to allow different datasets to be joined in an unambiguous manner. In addition, SPARQL provides specific graph traversal syntax for data that can be thought of as a graph. Some implementations, such as SPARQLverse also allow additional triple attributes such as timestamp and allow additional analytic functionality such as windowed aggregates.

SparQL queries RDF triplets in the subject-predicate-object form, using variables wherever we need output. The variables come with the question mark symbol (?) ahead. (E.g. ?x, ?y, ?temp). Every triplet in a single query is separated with a dot symbol (.) with the exception of the same subject, then we separate them with the semicolon (;) symbol.

Example:

```
select ?x where{ ex1:company ex1:hasEmployee ?x.?x ex1:employeeName
"JOHN" }
```

SparQL can also implement the class of an object in a query, using the letter (a) as a abbreviation of rdf: type. Another tool for easier queries is the blanc node implementation, where we supplement the subject or the object of the triplet with the (:blanc node) node. Like the owl files and the RuleML files, the SparQL can also use prefixes to replace the full URIs, which is really useful for the programmer and saves a lot of time. Some of the most used prefixes include:

```
PREFIX owl:<"http://www.w3.org/2002/07/owl#">
PREFIX xsd:<http://www.w3.org/2001/XMLSchema>
PREFIX rdfs:<"http://www.w3.org/2000/01/rdf-schema#">
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX time:<http://www.w3.org/2006/time#>
PREFIX time-entry <http://www.isi.edu/~hobbs/damltime/time-entry.owl#>
```

With SparQL the basic query rules are the same as with every query language. For example we can use the SELECT keyword to select a specific subset from which we want data to be extracted. As with SQL we can add distinct to this keyword to eliminate repeated results. If we want to specify the dataset of our query we use the FROM keyword or else it will query the default dataset. Finally the WHERE keyword can search inside triplets, filters, unions and optional path expressions and is optional. Filters are used also like with every other query language and can be logical (!, &&,||), mathematical expressions (+, -, *, /), comparison (<, >, =, !=) but also some specific for the semantic web such as SparQL tests(isURI, isBlanc, isLiteral, bound) or SparQL accessors (str, lang, datatype) and others. Finally SparQL uses Modifiers like “limit”, “order by” and “offset” to define the quantity of the results that we want to be extracted.

5 Methodology

Purpose of this thesis is the implementation of a time reasoner with the ability to extract new knowledge in RDF triplets from an ontology.

The first step is to create an RDF document. This document will be an ontology with an RDFS syntax. For the purpose of the project we will create a calendar type document, consisting of six (6) meetings. Those meetings are Interval Events, which means that they have a beginning, duration over time and an end. The document can be created in any text editor, but there is a better solution. Using Protégé, we can see the advantages of creating ontologies. Protégé has an easy to learn, hard to master user interface, consisting of Tabs. Because we already know that we will use the time-entry ontology as the base of our ontology, we already downloaded the (time-entry.owl) file and loaded it in the software. From there, we added the Individuals we needed, as seen below.

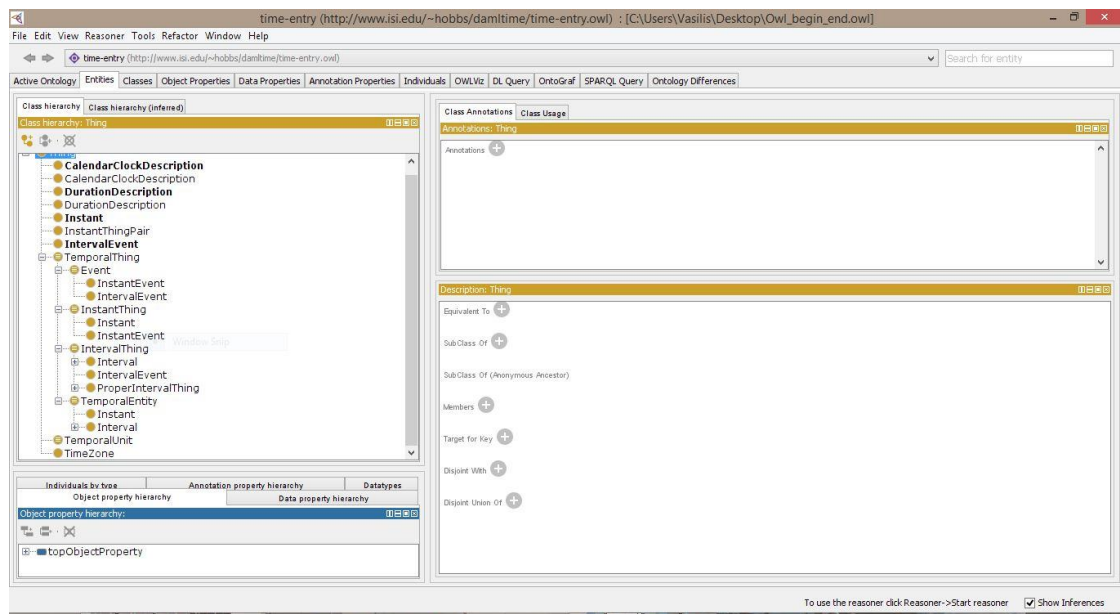


Figure 2 Entities Tab

In the Figure 2 above we can see our ontology as seen from the Entities Tab. On the left we see our Classes and on the right side we see the Annotations and Descriptions of every Class. Note that the top Class in every ontology is the Thing class and that cannot change.

Design and Implementation of a Time Reasoner for Knowledge Representation on RDFS

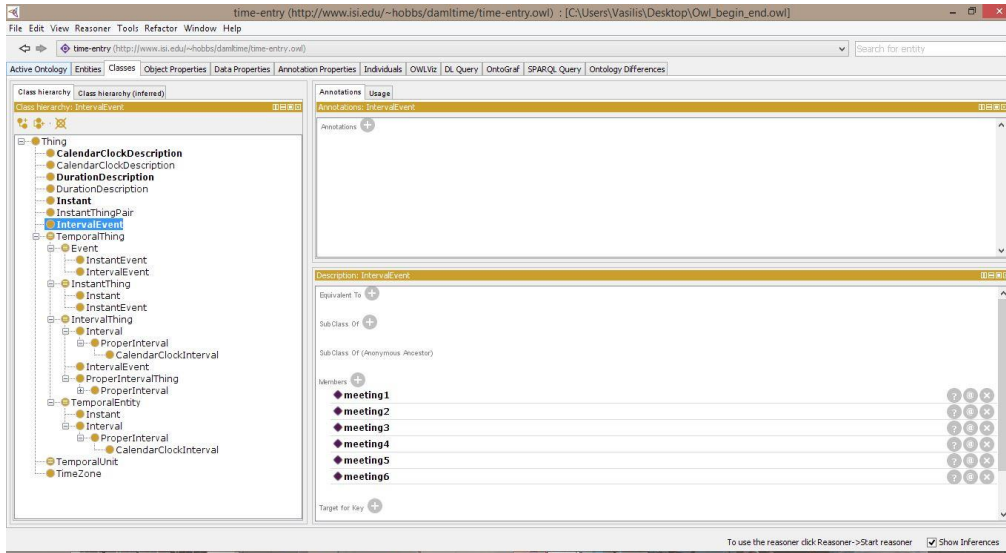


Figure 3 Classes Tab

In the Figure 3 we have selected the Interval Event Class and we can see the members of the class that we have created. As we can see there are six members, meeting1 to meeting6. If we selected the Instant class we would see twelve members, consisting of the beginnings and endings of every Interval Event. In the Duration Description class there are six members, the durations of every Interval Event. Finally the CalendarClockDescription has twelve members, consisting of the beginning and end description of every interval event.

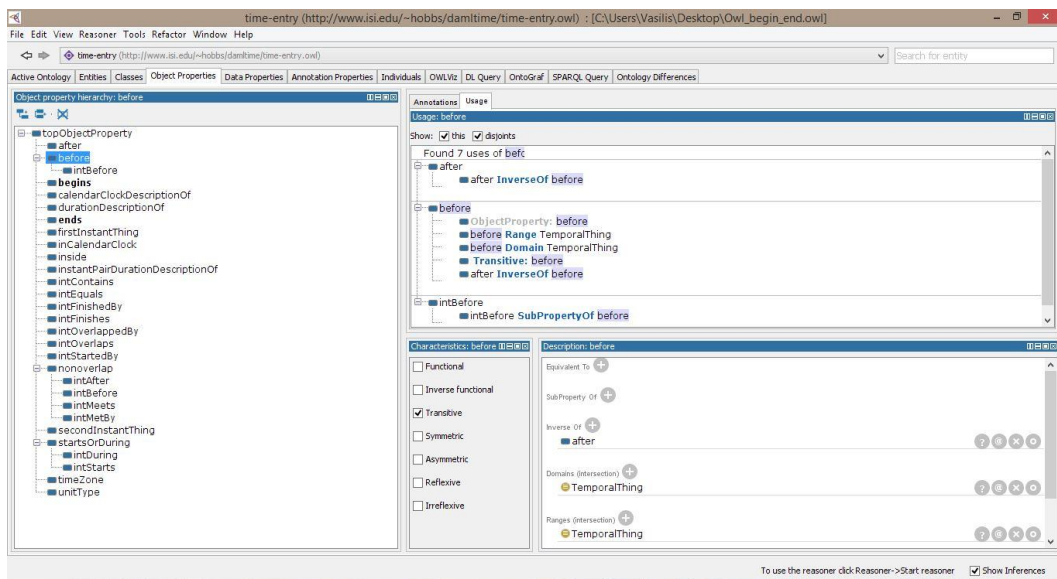


Figure 4 Object Properties Tab

OWL properties represent relationships and Object Properties are relationships between Individuals. Some object properties may have a corresponding inverse property. Properties may have a domain and a range, for example *before* has *TemporalThing* domain and range.

Finally, Protégé supports object properties characteristics like: Functional properties, Inverse Functional properties, Transitive property, Symmetric, Asymmetric, Reflexive, and Irreflexive.

Design and Implementation of a Time Reasoner for Knowledge Representation on RDFS

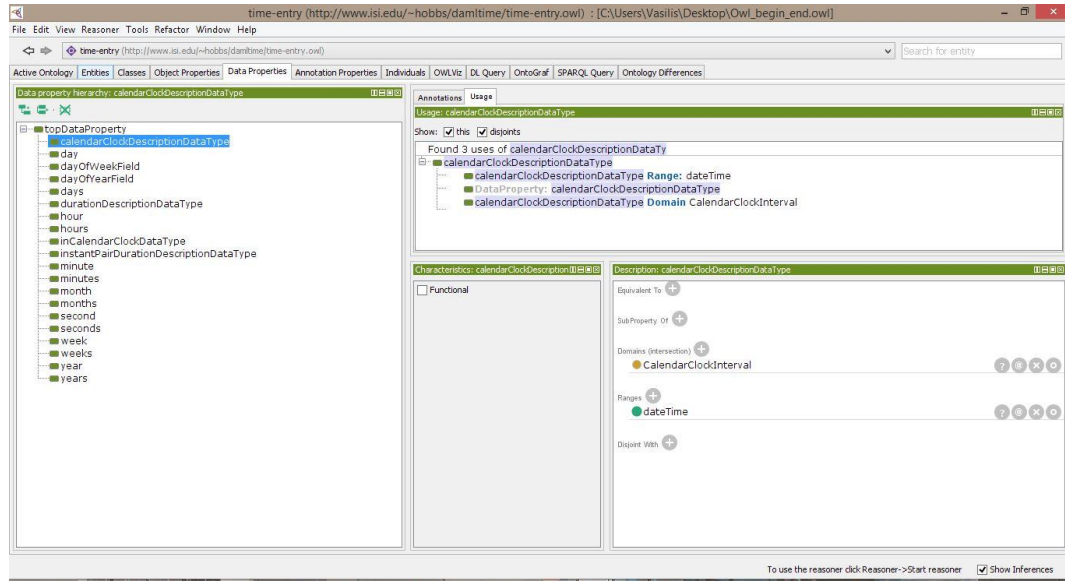


Figure 5 Data Properties Tab

Data type properties link Individuals to XML Schema datatypes or RDF literals. They describe relationships between individuals and data values but they can also be used as restrictions. Built-in datatypes are specified in the XML Schema vocabulary. In our ontology we have data types like day, hour, inCalendarClockDataType and others.

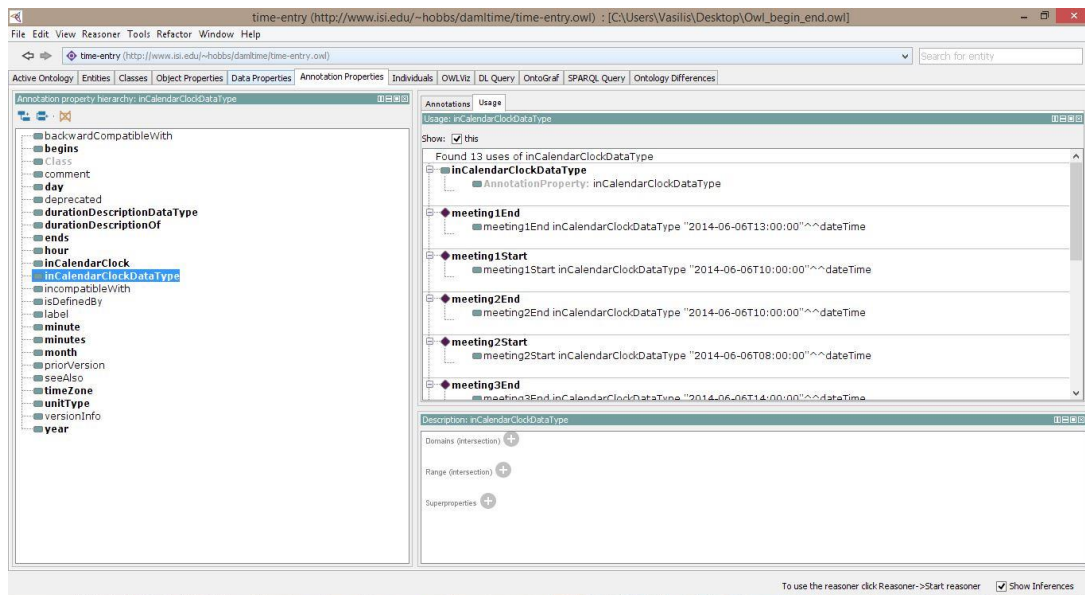


Figure 6 Annotation Properties Tab

Annotation properties are the third type of OWL properties. They are used to add information in the form of metadata to classes, Individuals, object or data type properties. We can see the usage above on figure 6.

Design and Implementation of a Time Reasoner for Knowledge Representation on RDFS

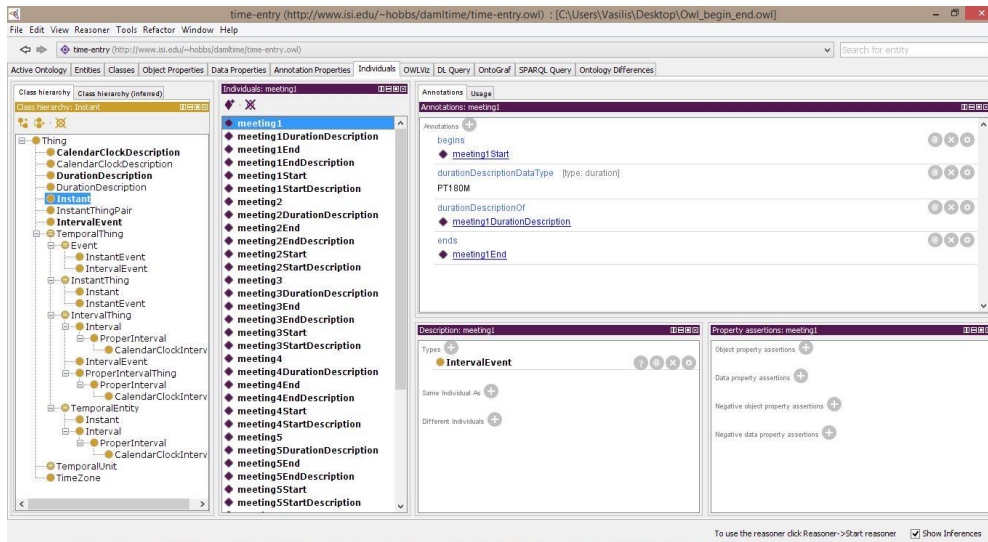


Figure 7 Individuals Tab

Above, on figure 7 we can see the Individuals Tab with the Interval Event meeting1 selected. We can spot the annotations on the right side and the description on the bottom. This view is the best for the user to be able to see the full characteristics of an Individual class.

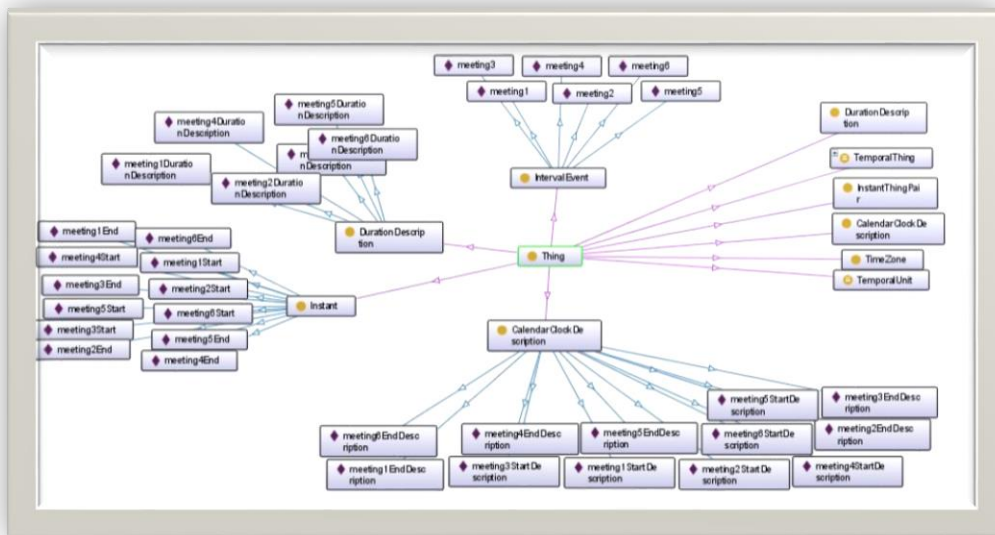


Figure 8 Graph Model

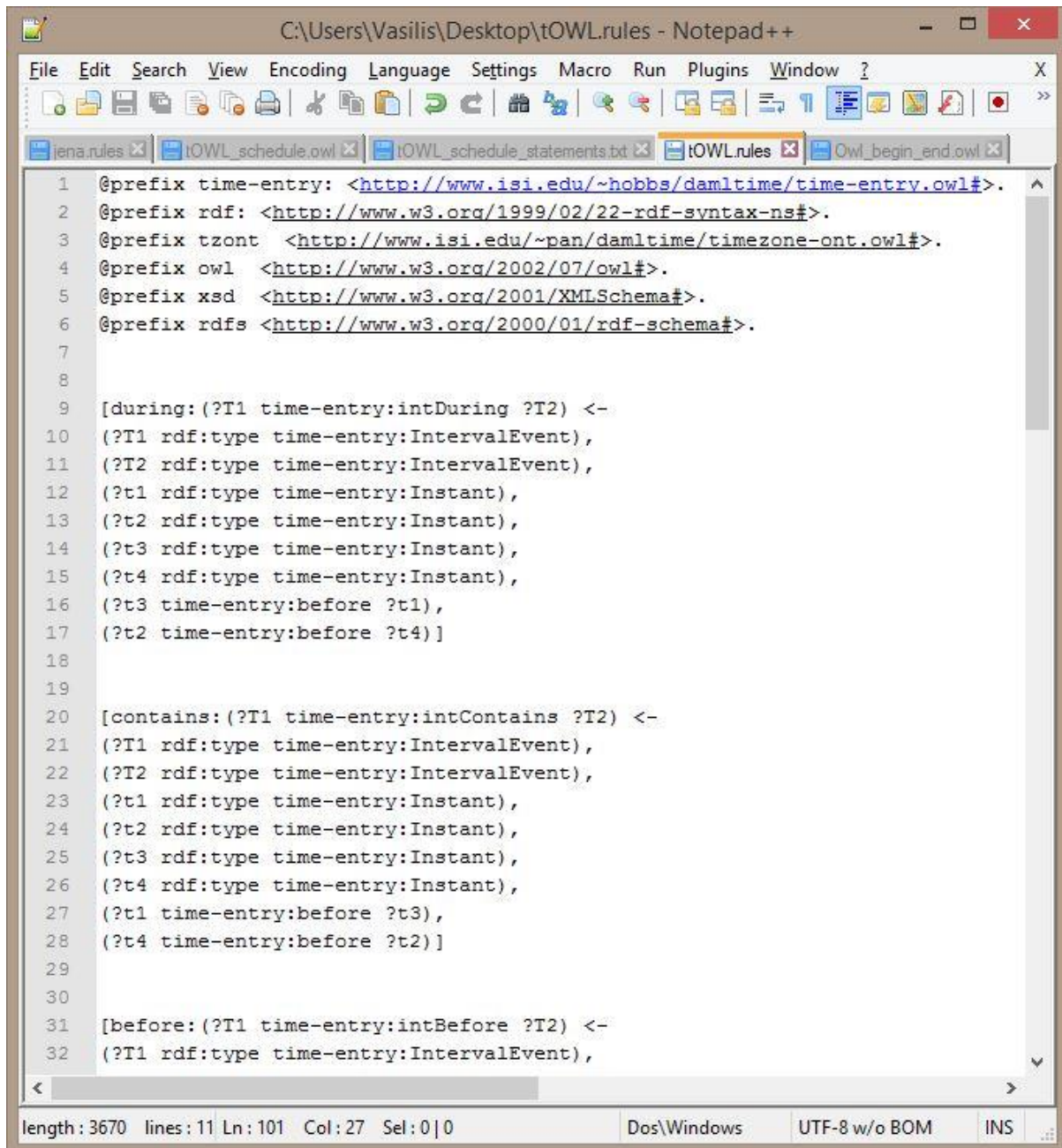
Above we see the graph model of the ontology. This is the easiest way to visualize an ontology and understand its components.

The ontology we created has Interval Events on the same day but with different time and durations. This was on purpose for this project to be able to show the capabilities of a reasoner, as it has to extract from the given data the knowledge of when one meeting event is chronically interchanged with another.

To check this hypothesis we have to know how many different time based relations are possible between two interval events at any time. The answer was given to us from a scientific paper called "Maintaining Knowledge about Temporal Intervals" written by James F. Allen in 1983. In this paper it was stated that there are thirteen possible relations between two events. For the First Order Logic (or Predicate logic) that proves those relations you can refer to the Appendix.

Design and Implementation of a Time Reasoner for Knowledge Representation on RDFS

The next step was to translate those definitions into RuleML syntax so the General purpose reasoner of the Jena API could read them. This was possible in any text editor, like Notepad++. First the prefixes were created and then the rules, most of which with the backward-chaining order.

The image shows a Notepad++ window titled "C:\Users\Vasilis\Desktop\tOWL.rules - Notepad++". The window contains several tabs, with "tOWL.rules" selected. The text in the editor is as follows:

```
1 @prefix time-entry: <http://www.isi.edu/~hobbs/damlttime/time-entry.owl#>.
2 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
3 @prefix tzont <http://www.isi.edu/~pan/damlttime/timezone-ont.owl#>.
4 @prefix owl <http://www.w3.org/2002/07/owl#>.
5 @prefix xsd <http://www.w3.org/2001/XMLSchema#>.
6 @prefix rdfs <http://www.w3.org/2000/01/rdf-schema#>.
7
8
9 [during:(?T1 time-entry:intDuring ?T2) <-
10 (?T1 rdf:type time-entry:IntervalEvent),
11 (?T2 rdf:type time-entry:IntervalEvent),
12 (?t1 rdf:type time-entry:Instant),
13 (?t2 rdf:type time-entry:Instant),
14 (?t3 rdf:type time-entry:Instant),
15 (?t4 rdf:type time-entry:Instant),
16 (?t3 time-entry:before ?t1),
17 (?t2 time-entry:before ?t4)]
18
19
20 [contains:(?T1 time-entry:intContains ?T2) <-
21 (?T1 rdf:type time-entry:IntervalEvent),
22 (?T2 rdf:type time-entry:IntervalEvent),
23 (?t1 rdf:type time-entry:Instant),
24 (?t2 rdf:type time-entry:Instant),
25 (?t3 rdf:type time-entry:Instant),
26 (?t4 rdf:type time-entry:Instant),
27 (?t1 time-entry:before ?t3),
28 (?t4 time-entry:before ?t2)]
29
30
31 [before:(?T1 time-entry:intBefore ?T2) <-
32 (?T1 rdf:type time-entry:IntervalEvent),
```

The status bar at the bottom shows "length: 3670 lines: 11 Ln: 101 Col: 27 Sel: 0|0", "Dos\Windows", "UTF-8 w/o BOM", and "INS".

Figure 9 Jena Rules file

The results are in the Appendix. After that was done we created a java program with the Jena API installed that could first load the ontology and then the custom rules we created. The java program was created in Eclipse using the Jena API. Finally it had to extract the statements in triplets along with the new knowledge.

6 Conclusions

As we can see, the possibilities are endless. Depending on the rules that we define, we can extract new knowledge in triplets. Despite that the purpose of this project was to infer about the interval relations between the Interval Events, we can see that even in a sub-ontology as the time-entry with very few predefined classes the amount of new information can be very impressive. Time entry and OWL Time can also support Time Zone defined classes.

For a web application oriented stand point, it is easy to see how important this new technology is. First of all, the ability to store time related data between objects can increase incrementally the odds of a search engine to give us more specific answers, instead of numerous insignificant results bases strictly on keyword search. Also the agents, used in the semantic web applications can use this technology to can use this technology to save the end user time.

This project shows just how important metadata manipulation and ontology creation is for the future of the web. With predefined ontologies that can save time and help better integrate the almost one billion web pages of the web with each other and reasoners that can extract new knowledge based on existed one, from returning better results to making decisions.

Design and Implementation of a Time Reasoner for Knowledge Representation on RDFS

Jena API Installation in Eclipse

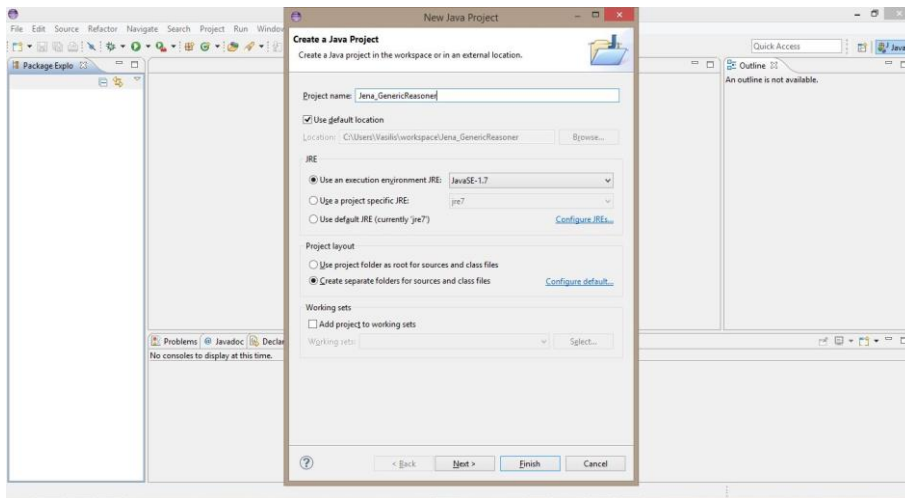


Figure 10 New Java Project

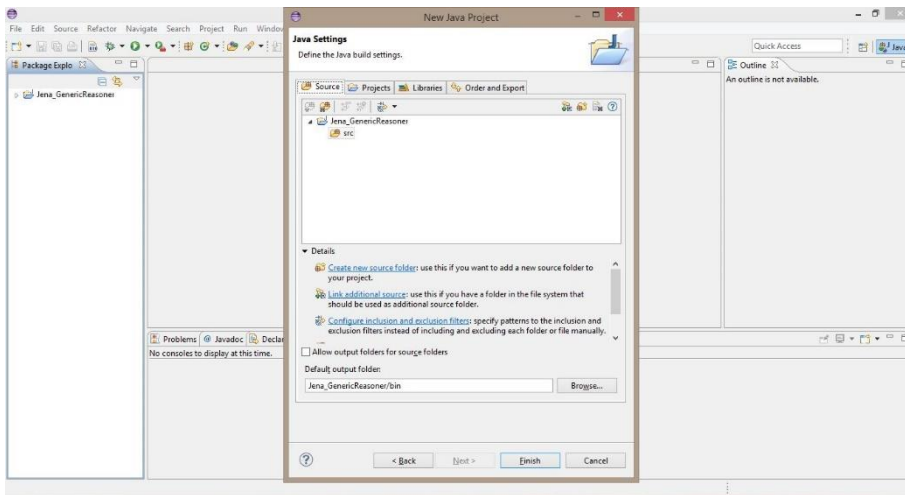


Figure 11 Project Name

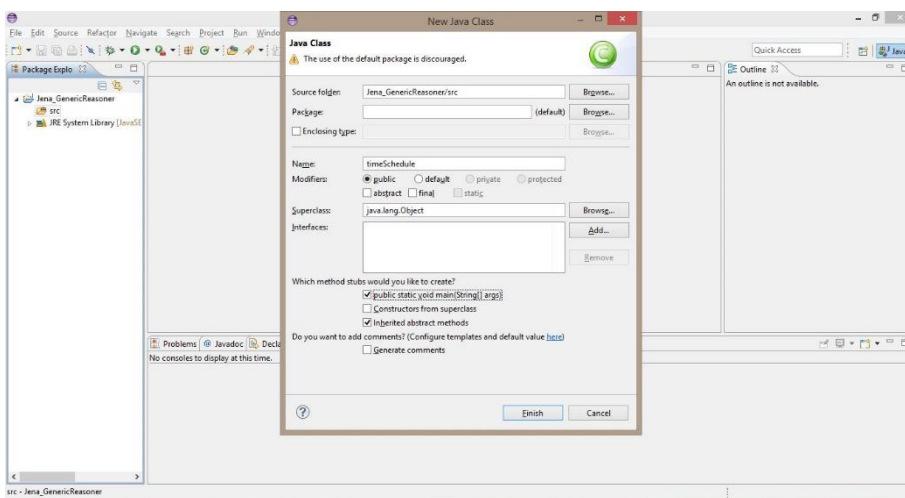


Figure 12 Create Java Class

Design and Implementation of a Time Reasoner for Knowledge Representation on RDFS

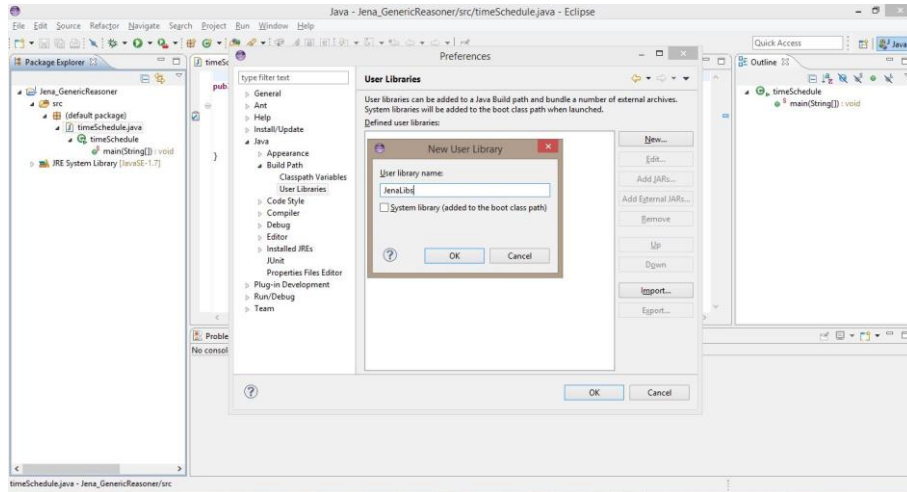


Figure 13 Create New User Library

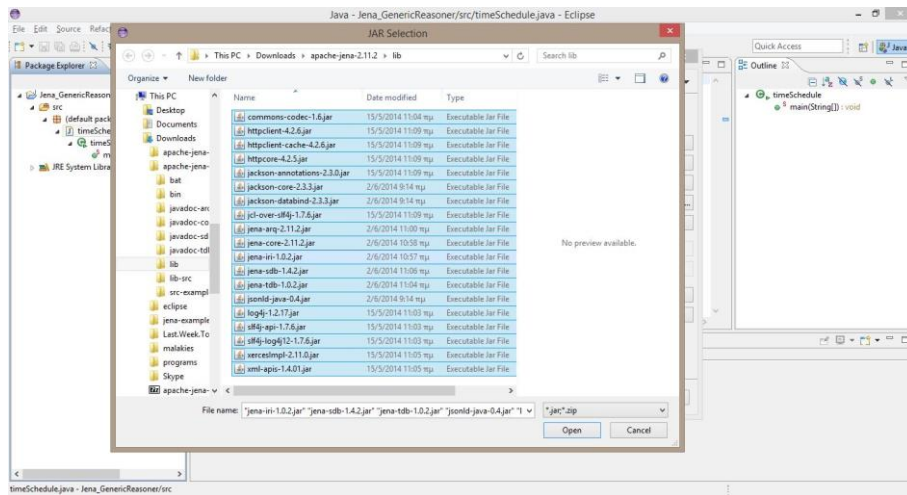


Figure 14 Insert Jar library files

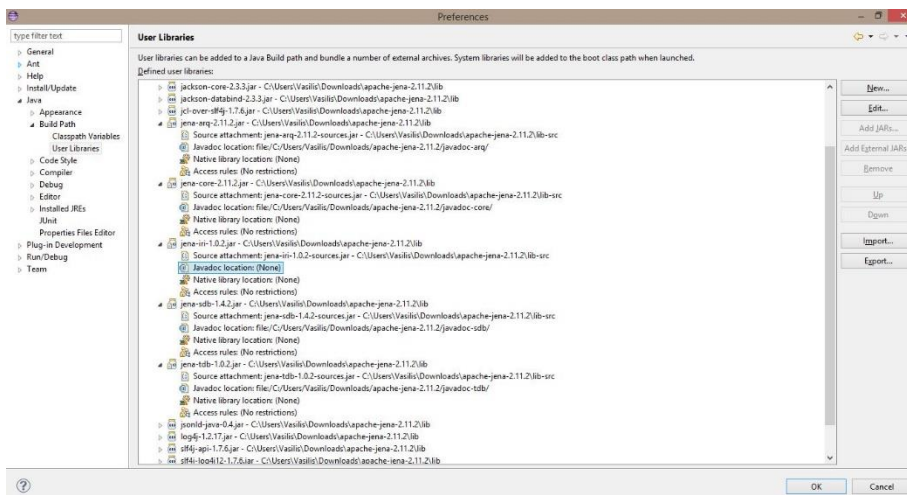


Figure 15 Define Javadoc and Sources

Design and Implementation of a Time Reasoner for Knowledge Representation on RDFS

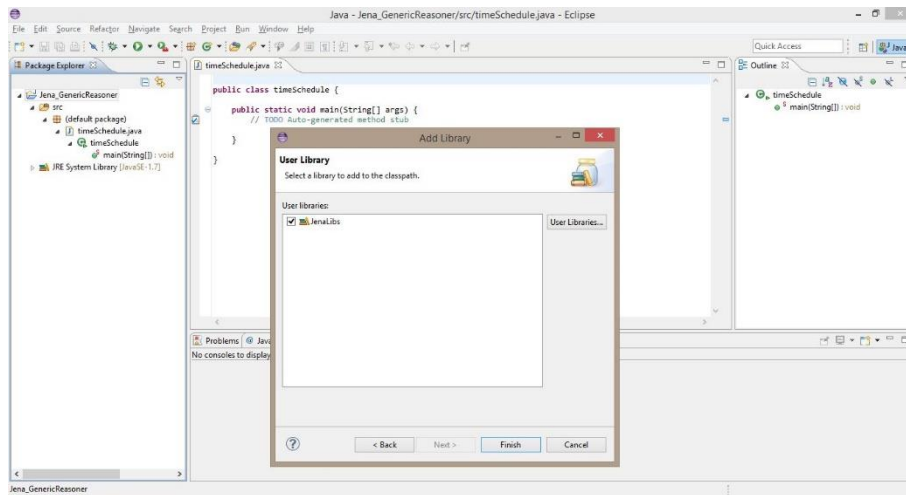


Figure 16 Import User Library Jena Libs

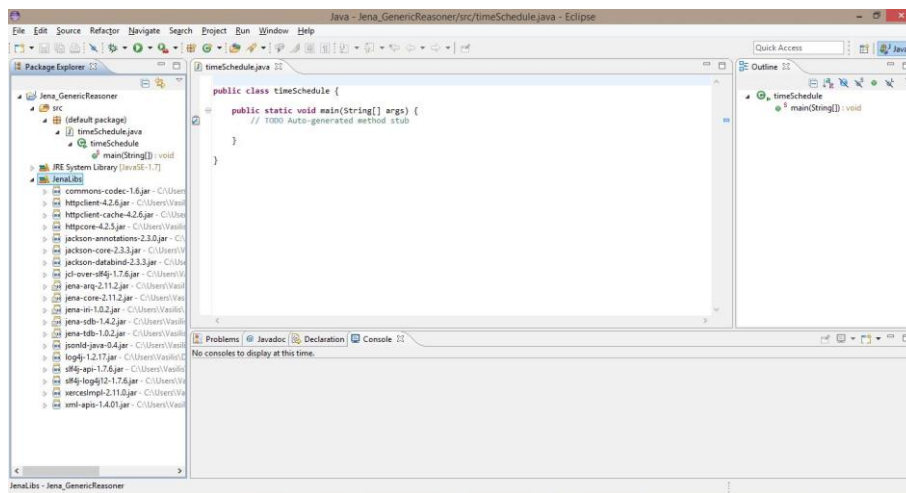


Figure 17 Final Results

First Order Logic Interval Relations

$$\begin{aligned}
 (\forall T_1, T_2)[\text{intEquals}(T_1, T_2)] \\
 &\equiv [\text{ProperInterval}(T_1) \wedge \text{ProperInterval}(T_2) \\
 &\quad \wedge (\forall t_1)[\text{begins}(t_1, T_1) \equiv \text{begins}(t_1, T_2)] \\
 &\quad \wedge (\forall t_2)[\text{ends}(t_2, T_1) \equiv \text{ends}(t_2, T_2)]]
 \end{aligned}$$

$$\begin{aligned}
 (\forall T_1, T_2)[\text{intBefore}(T_1, T_2)] \\
 &\equiv \text{ProperInterval}(T_1) \wedge \text{ProperInterval}(T_2) \wedge \text{before}(T_1, T_2)
 \end{aligned}$$

$$\begin{aligned}
 (\forall T_1, T_2)[\text{intMeets}(T_1, T_2)] \\
 &\equiv [\text{ProperInterval}(T_1) \wedge \text{ProperInterval}(T_2) \\
 &\quad \wedge (\exists t)[\text{ends}(t, T_1) \wedge \text{begins}(t, T_2)]]
 \end{aligned}$$

$$\begin{aligned}
 (\forall T_1, T_2)[\text{intOverlaps}(T_1, T_2)] \\
 &\equiv [\text{ProperInterval}(T_1) \wedge \text{ProperInterval}(T_2) \\
 &\quad \wedge (\exists t_2, t_3)[\text{ends}(t_2, T_1) \wedge \text{begins}(t_3, T_2) \wedge \text{before}(t_3, t_2)] \\
 &\quad \wedge (\forall t_1)[\text{begins}(t_1, T_1) \supset \text{before}(t_1, t_3)] \\
 &\quad \wedge (\forall t_4)[\text{ends}(t_4, T_2) \supset \text{before}(t_2, t_4)]]
 \end{aligned}$$

$$\begin{aligned}
 (\forall T_1, T_2)[\text{intStarts}(T_1, T_2)] \\
 &\equiv [\text{ProperInterval}(T_1) \wedge \text{ProperInterval}(T_2) \\
 &\quad \wedge (\exists t_2)[\text{ends}(t_2, T_1) \wedge (\forall t_1)[\text{begins}(t_1, T_1) \equiv \text{begins}(t_1, T_2)]] \\
 &\quad \wedge (\forall t_4)[\text{ends}(t_4, T_2) \supset \text{before}(t_2, t_4)]]
 \end{aligned}$$

$$\begin{aligned}
 (\forall T_1, T_2)[\text{intDuring}(T_1, T_2)] \\
 &\equiv [\text{ProperInterval}(T_1) \wedge \text{ProperInterval}(T_2) \\
 &\quad \wedge (\exists t_1, t_2)[\text{begins}(t_1, T_1) \wedge \text{ends}(t_2, T_1) \\
 &\quad \wedge (\forall t_3)[\text{begins}(t_3, T_2) \supset \text{before}(t_3, t_1)] \\
 &\quad \wedge (\forall t_4)[\text{ends}(t_4, T_2) \supset \text{before}(t_2, t_4)]]
 \end{aligned}$$

$$\begin{aligned}
 (\forall T_1, T_2)[\text{intFinishes}(T_1, T_2)] \\
 &\equiv [\text{ProperInterval}(T_1) \wedge \text{ProperInterval}(T_2) \\
 &\quad \wedge (\exists t_1)[\text{begins}(t_1, T_1) \wedge (\forall t_3)[\text{begins}(t_3, T_2) \supset \text{before}(t_3, t_1)]] \\
 &\quad \wedge (\forall t_4)[\text{ends}(t_4, T_2) \equiv \text{ends}(t_4, T_1)]]
 \end{aligned}$$

$$\text{intAfter}(T_1, T_2) \equiv \text{intBefore}(T_2, T_1)$$

$$\text{intMetBy}(T_1, T_2) \equiv \text{intMeets}(T_2, T_1)$$

$$\text{intOverlappedBy}(T_1, T_2) \equiv \text{intOverlaps}(T_2, T_1)$$

$$\text{intStartedBy}(T_1, T_2) \equiv \text{intStarts}(T_2, T_1)$$

$$\text{intContains}(T_1, T_2) \equiv \text{intDuring}(T_2, T_1)$$

$$\text{intFinishedBy}(T_1, T_2) \equiv \text{intFinishes}(T_2, T_1)$$

Bibliography

1. Feng Pan, Jerry R. Hobbs, *Temporal Aggregates in OWL-Time*, Information Sciences Institute, University of Southern California, 2005
2. Feng Pan, Jerry R. Hobbs, *Time in OWL-S*, Information Sciences Institute, University of Southern California, 2004
3. Jerry R. Hobbs, James Pustejovsky, *Annotating and Reasoning about Time and Events*, University of Southern California, Brandeis University, 2003
4. Chris WELTY, Richard FIKES, *A Reusable Ontology for Fluents in OWL*, IBM Watson Research Center, USA, Stanford AI Lab - Knowledge Systems, USA, 2006
5. JAMES F. ALLEN, *Maintaining Knowledge about Temporal Intervals*, The University of Rochester, 1983
6. Sotiris Batsakis, Euripides G.M. Petrakis, *Representing Temporal Knowledge in the Semantic Web: The Extended 4D Fluents Approach*, Department of Electronic and Computer Engineering, Technical University of Crete (TUC), 2010
7. Nikos Papadakis, Dimitris Plexousakis, *Actions with Duration and Constraints: the Ramification Problem in Temporal Databases*, Department of Computer Science University of Crete and Institute of Computer Science FORTH Greece, 2003
8. Steven Schockaert, Martine De Cock, and Etienne E. Kerre, *Fuzzifying Allen's Temporal Interval Relations*, 2008
9. Viorel Milea, Flavius Frasincar, and Uzay Kaymak, *A Temporal Web Ontology Language*, Erasmus Research Institute of Management (ERIM), RSM Erasmus University / Erasmus School of Economics, Erasmus Universiteit Rotterdam, September 2009

References

<http://www.ics.uci.edu/~alspaugh/cls/shr/allen.html>
[http://en.wikipedia.org/wiki/Fluent_\(artificial_intelligence\)](http://en.wikipedia.org/wiki/Fluent_(artificial_intelligence))
<http://jena.apache.org/documentation/>
<http://owl.cs.manchester.ac.uk/research/co-ode/>
<http://wenku.baidu.com/view/39a29416866fb84ae45c8ddd.html>
<https://wiki.csc.calpoly.edu/>
<http://www.slideshare.net/>
<http://hydrogen.informatik.tu-cottbus.de/wiki/index.php/JenaRules>
<http://www.ibm.com/developerworks/java/library/j-jena/index.html>
<http://staff.um.edu.mt/cabe2/lectures/webscience/docs/jena.pdf>
<http://owl.cs.manchester.ac.uk/research/co-ode/>
<http://webdam.inria.fr/Jorge/html/wdmch9.html#x14-1830008>
<http://docs.openlinksw.com/virtuoso/rdfsparqlrule.html>
<http://www.w3.org/TR/owl-time/>
<http://www.isi.edu/~hobbs/owl-time.html>
<http://en.wikipedia.org/wiki/RDFS>

Appendix

RuleSet:

```
@prefix time-entry: <http://www.isi.edu/~hobbs/damlltime/time-
entry.owl#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix tzont <http://www.isi.edu/~pan/damlltime/timezone-ont.owl#>.
@prefix owl <http://www.w3.org/2002/07/owl#>.
@prefix xsd <http://www.w3.org/2001/XMLSchema#>.
@prefix rdfs <http://www.w3.org/2000/01/rdf-schema#>.
```

```
[during:(?T1 time-entry:intDuring ?T2) <-
(?T1 rdf:type time-entry:IntervalEvent),
(?T2 rdf:type time-entry:IntervalEvent),
(?t1 rdf:type time-entry:Instant),
(?t2 rdf:type time-entry:Instant),
(?t3 rdf:type time-entry:Instant),
(?t4 rdf:type time-entry:Instant),
(?t3 time-entry:before ?t1),
(?t2 time-entry:before ?t4)]
```

```
[contains:(?T1 time-entry:intContains ?T2) <-
(?T1 rdf:type time-entry:IntervalEvent),
(?T2 rdf:type time-entry:IntervalEvent),
(?t1 rdf:type time-entry:Instant),
(?t2 rdf:type time-entry:Instant),
(?t3 rdf:type time-entry:Instant),
(?t4 rdf:type time-entry:Instant),
(?t1 time-entry:before ?t3),
(?t4 time-entry:before ?t2)]
```

```
[before:(?T1 time-entry:intBefore ?T2) <-
(?T1 rdf:type time-entry:IntervalEvent),
(?T2 rdf:type time-entry:IntervalEvent),
(?t1 time-entry:before ?t3),
(?t2 time-entry:before ?t3)]
```

```
[after:(?T1 time-entry:intAfter ?T2) <-
(?T1 rdf:type time-entry:IntervalEvent),
(?T2 rdf:type time-entry:IntervalEvent),
(?t3 time-entry:before ?t1),
(?t4 time-entry:before ?t1)]
```

```
[overlaps:(?T1 time-entry:intOverlaps ?T2) <-
```

```
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),  
(?t1 time-entry:begins ?T1),  
(?t2 time-entry:ends ?T1),  
(?t3 time-entry:begins ?T2),  
(?t4 time-entry:ends ?T2),  
(?t1 time-entry:before ?t3),  
(?t3 time-entry:before ?t2),  
(?t2 time-entry:before ?t4)]
```

```
[overlappedby:(?T1 time-entry:intOverlappedBy ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),  
(?t1 time-entry:begins ?T1),  
(?t2 time-entry:ends ?T1),  
(?t3 time-entry:begins ?T2),  
(?t4 time-entry:ends ?T2),  
(?t3 time-entry:before ?t1),  
(?t1 time-entry:before ?t4),  
(?t4 time-entry:before ?t2)]
```

```
[equals:(?T1 time-entry:intEquals ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),  
(?t1 time-entry:begins ?T1),  
(?t2 time-entry:ends ?T1),  
(?t1 time-entry:begins ?T2),  
(?t2 time-entry:ends ?T2)]
```

```
[starts:(?T1 time-entry:intStarts ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),  
(?t1 time-entry:before ?t4),  
(?t2 time-entry:before ?t4),  
(?t3 time-entry:before ?t4)]
```

```
[startedby:(?T1 time-entry:intStartedBy ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),  
(?t1 time-entry:before ?t4),  
(?t2 time-entry:after ?t4),  
(?t3 time-entry:before ?t4)]
```

```
[finishes:(?T1 time-entry:intFinishes ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),
```

```
(?t3 time-entry:before ?t1),  
(?t2 time-entry:ends ?T1),  
(?t3 time-entry:before ?t1)  
(?t4 time-entry:ends ?T2)]
```

```
[finishedby:(?T1 time-entry:intFinishedBy ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),  
(?t1 time-entry:before ?t3),  
(?t2 time-entry:ends ?T1),  
(?t3 time-entry:after ?t1)  
(?t4 time-entry:ends ?T2)]
```

```
[meets:(?T1 time-entry:intMeets ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),  
(?t time-entry:ends ?T1),  
(?t time-entry:begins ?T2)]
```

```
[metby:(?T1 time-entry:intMetBy ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),  
(?t time-entry:begins ?T2),  
(?t time-entry:ends ?T1)]
```

Ontology snippet:

```
<!-- http://www.isi.edu/~hobbs/damlttime/time-entry.owl#meeting1 -->

<owl:NamedIndividual rdf:about="&time-entry;meeting1">
  <rdf:type rdf:resource="&time-entry;IntervalEvent"/>
  <durationDescriptionDataType
rdf:datatype="&xsd;duration">PT180M</durationDescriptionDataType>
  <durationDescriptionOf rdf:resource="&time-entry;meeting1DurationDescription"/>
  <ends rdf:resource="&time-entry;meeting1End"/>
  <begins rdf:resource="&time-entry;meeting1Start"/>
</owl:NamedIndividual>

<!-- http://www.isi.edu/~hobbs/damlttime/time-entry.owl#meeting1DurationDescription -->

<owl:NamedIndividual rdf:about="&time-entry;meeting1DurationDescription">
  <rdf:type rdf:resource="&time-entry;DurationDescription"/>
  <minutes rdf:datatype="&xsd;decimal">180</minutes>
</owl:NamedIndividual>

<!-- http://www.isi.edu/~hobbs/damlttime/time-entry.owl#meeting1End -->

<owl:NamedIndividual rdf:about="&time-entry;meeting1End">
  <rdf:type rdf:resource="&time-entry;Instant"/>
  <inCalendarClockDataType rdf:datatype="&xsd;dateTime">2014-06-
06T13:00:00</inCalendarClockDataType>
  <inCalendarClock rdf:resource="&time-entry;meeting1StartDescription"/>
</owl:NamedIndividual>

<!-- http://www.isi.edu/~hobbs/damlttime/time-entry.owl#meeting1EndDescription -->

<owl:NamedIndividual rdf:about="&time-entry;meeting1EndDescription">
  <rdf:type rdf:resource="&time-entry;CalendarClockDescription"/>
  <minute rdf:datatype="&xsd;nonNegativeInteger">0</minute>
  <hour rdf:datatype="&xsd;nonNegativeInteger">13</hour>
  <year rdf:datatype="&xsd;gYear">2014</year>
  <day rdf:datatype="&xsd;gDay">6</day>
  <month rdf:datatype="&xsd;gMonth">6</month>
  <unitType rdf:resource="&time-entry;unitMinute"/>
  <timeZone rdf:resource="http://www.isi.edu/~hobbs/damlttime/timezone-
world.owl#BTZ"/>
</owl:NamedIndividual>
```



```
<!-- http://www.isi.edu/~hobbs/damlttime/time-entry.owl#meeting1Start -->

<owl:NamedIndividual rdf:about="&time-entry;meeting1Start">
  <rdf:type rdf:resource="&time-entry;Instant"/>
  <inCalendarClockDataType rdf:datatype="&xsd;dateTime">2014-06-
06T10:00:00</inCalendarClockDataType>
  <inCalendarClock rdf:resource="&time-entry;meeting1StartDescription"/>
</owl:NamedIndividual>

<!-- http://www.isi.edu/~hobbs/damlttime/time-entry.owl#meeting1StartDescription -->

<owl:NamedIndividual rdf:about="&time-entry;meeting1StartDescription">
  <rdf:type rdf:resource="&time-entry;CalendarClockDescription"/>
  <minute rdf:datatype="&xsd;nonNegativeInteger">0</minute>
  <hour rdf:datatype="&xsd;nonNegativeInteger">10</hour>
  <year rdf:datatype="&xsd;gYear">2014</year>
  <day rdf:datatype="&xsd;gDay">6</day>
  <month rdf:datatype="&xsd;gMonth">6</month>
  <unitType rdf:resource="&time-entry;unitMinute"/>
  <timeZone rdf:resource="http://www.isi.edu/~hobbs/damlttime/timezone-
world.owl#BTZ"/>
</owl:NamedIndividual>
```

Java Code:

```

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.List;

import com.hp.hpl.jena.rdf.model.InfModel;
import com.hp.hpl.jena.rdf.model.Literal;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.rdf.model.ModelFactory;
import com.hp.hpl.jena.rdf.model.Property;
import com.hp.hpl.jena.rdf.model.RDFNode;
import com.hp.hpl.jena.rdf.model.Resource;
import com.hp.hpl.jena.rdf.model.Statement;
import com.hp.hpl.jena.rdf.model.StmtIterator;
import com.hp.hpl.jena.reasoner.Reasoner;
import com.hp.hpl.jena.reasoner.ReasonerRegistry;
import com.hp.hpl.jena.reasoner.rulesys.GenericRuleReasoner;
import com.hp.hpl.jena.reasoner.rulesys.Rule;
import com.hp.hpl.jena.shared.DoesNotExistException;

public class timeSchedule {
    public static Model m;
    public static BufferedReader br;

    public static void main(String[] args) throws IOException {
        m = ModelFactory.createDefaultModel();

        String in = "";
        while (!in.equals("Q")) {
            try
            {
                in = getUserInput();
            }
            catch (DoesNotExistException e) {
                in = "";
            }
            execute(in);
        }
    }

    public static String getUserInput() throws IOException {
        br = new BufferedReader(new InputStreamReader(System.in));
        String input = null;

        System.out.println("Please enter a command");
        System.out.println("[1] Load model");
        System.out.println("[2] Run rules");
        System.out.println("[3] Print all statements");
        System.out.println("[4] Query model");
        System.out.println("[5] Print number of statements");
        System.out.println("[Q] Quit");

        input = br.readLine();
    }
}

```

```

        return input;
    }

    public static void execute(String command) throws IOException {
        if (command.equals("1")) {
            System.out.print("Enter model location: ");
            String input = br.readLine();
            File f = new File(input);
            if (f.exists() && f.isFile())
                m.read("file:" + input);
            else
                System.out.println("Bad file location");
        }
        else if (command.equals("2")) {
            System.out.print("Enter rules location: ");
            String input = br.readLine();
            if (input == null)
                return;
            File f = new File(input);
            if (f.exists()) {
                List<Rule> rules = Rule.rulesFromURL("file:" +
input);
                GenericRuleReasoner r = new
GenericRuleReasoner(rules);
                r.setOWLTranslation(true);
                r.setTransitiveClosureCaching(true);

                InfModel infmodel = ModelFactory.createInfModel(r,
m);
                m.add(infmodel.getDeductionsModel());
            }
            else
                System.out.println("That rules file does not
exist.");
        }
        else if (command.equals("3")) {
            StmtIterator si = m.listStatements();
            Statement s = null;
            while(si.hasNext()) {
                s = si.next();
                System.out.println(s);
            }
        }
        else if (command.equals("4")) {
            System.out.print("Enter a pattern to match: ");
            String input = br.readLine();
            String[] pattern = input.split(" ");
            if (pattern.length != 3) {
                System.out.println("Bad query pattern");
                return;
            }
            Resource s = null;
            Property p = null;
            RDFNode o = null;

```

```

if (pattern[0].matches("'.'+'"))
    s = getAnonNode(pattern[0].replace("'", ""));
else if (!pattern[0].equals("?"))
    s = m.getResource(pattern[0]);

    if (pattern[1].matches("'.'+'"))
        p = getAnonNode(pattern[1].replace("'", ""),
"").as(Property.class);
else if (!pattern[1].equals("?"))
    p = m.getProperty(pattern[1]);

    if (pattern[2].matches("'.'+'"))
        o = getAnonNode(pattern[2].replace("'", ""));
else if (pattern[2].matches("\\.+\\\\"))
        o = m.createLiteral(pattern[2].replace("\\", ""));
else if (!pattern[2].equals("?"))
        o = m.getResource(pattern[2]);

    StmtIterator si = m.listStatements(s, p, o);
    Statement st = null;
    while(si.hasNext()) {
        st = si.next();
        System.out.println(st);
    }
}
else if (command.equals("5")) {
    System.out.println(m.size());
}
}

private static Resource getAnonNode(String anonId) {
    StmtIterator si = m.listStatements();
    Statement s = null;
    while(si.hasNext()) {
        s = si.next();
        Resource node = s.getSubject();
        if (node.isAnon() && node.getId().toString().equals(anonId)) {
            return node;
        }
        node = s.getPredicate();
        if (node.isAnon() && node.getId().toString().equals(anonId)) {
            return node;
        }
        if (s.getObject().canAs(Resource.class)) {
            node = s.getObject().as(Resource.class);
            if (node.isAnon() &&
node.getId().toString().equals(anonId)) {
                return node;
            }
        }
    }
}
return null;
}
}

```

[Rule File \(tOwl.rules\)](#)

```
@prefix time-entry: <http://www.isi.edu/~hobbs/damlltime/time-entry.owl#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix tzont <http://www.isi.edu/~pan/damlltime/timezone-ont.owl#>.
@prefix owl <http://www.w3.org/2002/07/owl#>.
@prefix xsd <http://www.w3.org/2001/XMLSchema#>.
@prefix rdfs <http://www.w3.org/2000/01/rdf-schema#>.
```

[before:

```
(?x time-entry:before ?y) <-
(?x rdf:type time-entry:Instant),
(?y rdf:type time-entry:Instant),
(?x time-entry:inCalendarClockDatatype ?x1),
(?x1 time-entry:hour ?hour1),
(?y time-entry:inCalendarClockDatatype ?y1),
(?y1 time-entry:hour ?hour2),
lessThan(?hour1,?hour2)
]
```

[during:

```
(?T1 time-entry:intDuring ?T2) <-
(?T1 rdf:type time-entry:IntervalEvent),
(?T2 rdf:type time-entry:IntervalEvent),
(?t1 rdf:type time-entry:Instant),
(?t2 rdf:type time-entry:Instant),
(?t3 rdf:type time-entry:Instant),
(?t4 rdf:type time-entry:Instant),
(?T1 time-entry:begins ?t1),
(?T1 time-entry:ends ?t2),
(?T2 time-entry:begins ?t3),
(?T2 time-entry:ends ?t4),
(?t3 time-entry:before ?t1),
(?t2 time-entry:before ?t4)]
```

[contains:(?T1 time-entry:intContains ?T2) <-

```
(?T1 rdf:type time-entry:IntervalEvent),
(?T2 rdf:type time-entry:IntervalEvent),
(?t1 rdf:type time-entry:Instant),
(?t2 rdf:type time-entry:Instant),
(?t3 rdf:type time-entry:Instant),
(?t4 rdf:type time-entry:Instant),
(?T1 time-entry:begins ?t1),
(?T1 time-entry:ends ?t2),
(?T2 time-entry:begins ?t3),
```

```
(?T2 time-entry:ends ?t4),  
(?t1 time-entry:before ?t3),  
(?t4 time-entry:before ?t2)]
```

```
[intBefore:(?T1 time-entry:intBefore ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),  
(?t1 rdf:type time-entry:Instant),  
(?t2 rdf:type time-entry:Instant),  
(?t3 rdf:type time-entry:Instant),  
(?t4 rdf:type time-entry:Instant),  
(?T1 time-entry:begins ?t1),  
(?T1 time-entry:ends ?t2),  
(?T2 time-entry:begins ?t3),  
(?T2 time-entry:ends ?t4),  
(?t1 time-entry:before ?t3),  
(?t2 time-entry:before ?t3)]
```

```
[after:(?T1 time-entry:intAfter ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),  
(?t1 rdf:type time-entry:Instant),  
(?t2 rdf:type time-entry:Instant),  
(?t3 rdf:type time-entry:Instant),  
(?t4 rdf:type time-entry:Instant),  
(?T1 time-entry:begins ?t1),  
(?T1 time-entry:ends ?t2),  
(?T2 time-entry:begins ?t3),  
(?T2 time-entry:ends ?t4),  
(?t3 time-entry:before ?t1),  
(?t4 time-entry:before ?t1)]
```

```
[overlaps:(?T1 time-entry:intOverlaps ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),  
(?t1 rdf:type time-entry:Instant),  
(?t2 rdf:type time-entry:Instant),  
(?t3 rdf:type time-entry:Instant),  
(?t4 rdf:type time-entry:Instant),  
(?T1 time-entry:begins ?t1),  
(?T1 time-entry:ends ?t2),  
(?T2 time-entry:begins ?t3),  
(?T2 time-entry:ends ?t4),  
(?t1 time-entry:before ?t3),  
(?t3 time-entry:before ?t2),  
(?t2 time-entry:before ?t4)]
```

```
[overlappedby:(?T1 time-entry:intOverlappedBy ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),  
(?t1 rdf:type time-entry:Instant),  
(?t2 rdf:type time-entry:Instant),  
(?t3 rdf:type time-entry:Instant),  
(?t4 rdf:type time-entry:Instant),  
(?T1 time-entry:begins ?t1),  
(?T1 time-entry:ends ?t2),  
(?T2 time-entry:begins ?t3),  
(?T2 time-entry:ends ?t4),  
(?t3 time-entry:before ?t1),  
(?t1 time-entry:before ?t4),  
(?t4 time-entry:before ?t2)]
```

```
[equals:(?T1 time-entry:intEquals ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),  
(?t1 rdf:type time-entry:Instant),  
(?t2 rdf:type time-entry:Instant),  
(?t3 rdf:type time-entry:Instant),  
(?t4 rdf:type time-entry:Instant),  
(?T1 time-entry:begins ?t1),  
(?T1 time-entry:ends ?t2),  
(?T2 time-entry:begins ?t3),  
(?T2 time-entry:ends ?t4),]
```

```
[starts:(?T1 time-entry:intStarts ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),  
(?t1 rdf:type time-entry:Instant),  
(?t2 rdf:type time-entry:Instant),  
(?t3 rdf:type time-entry:Instant),  
(?t4 rdf:type time-entry:Instant),  
(?T1 time-entry:begins ?t1),  
(?T1 time-entry:ends ?t2),  
(?T2 time-entry:begins ?t3),  
(?T2 time-entry:ends ?t4),  
(?t1 time-entry:before ?t4),  
(?t2 time-entry:before ?t4),  
(?t3 time-entry:before ?t4)]
```

```
[startedby:(?T1 time-entry:intStartedBy ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),
```

```
(?t1 rdf:type time-entry:Instant),  
(?t2 rdf:type time-entry:Instant),  
(?t3 rdf:type time-entry:Instant),  
(?t4 rdf:type time-entry:Instant),  
(?T1 time-entry:begins ?t1),  
(?T1 time-entry:ends ?t2),  
(?T2 time-entry:begins ?t3),  
(?T2 time-entry:ends ?t4),  
(?t1 time-entry:before ?t4),  
(?t2 time-entry:after ?t4),  
(?t3 time-entry:before ?t4)]
```

```
[finishes:(?T1 time-entry:intFinishes ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),  
(?t1 rdf:type time-entry:Instant),  
(?t2 rdf:type time-entry:Instant),  
(?t3 rdf:type time-entry:Instant),  
(?t4 rdf:type time-entry:Instant),  
(?T1 time-entry:begins ?t1),  
(?T1 time-entry:ends ?t2),  
(?T2 time-entry:begins ?t3),  
(?T2 time-entry:ends ?t4),  
(?t3 time-entry:before ?t1),  
(?t2 time-entry:ends ?T1),  
(?t3 time-entry:before ?t1)  
(?t4 time-entry:ends ?T2)]
```

```
[finishedby:(?T1 time-entry:intFinishedBy ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),  
(?t1 rdf:type time-entry:Instant),  
(?t2 rdf:type time-entry:Instant),  
(?t3 rdf:type time-entry:Instant),  
(?t4 rdf:type time-entry:Instant),  
(?T1 time-entry:begins ?t1),  
(?T1 time-entry:ends ?t2),  
(?T2 time-entry:begins ?t3),  
(?T2 time-entry:ends ?t4),  
(?t1 time-entry:before ?t3),  
(?t2 time-entry:ends ?T1),  
(?t3 time-entry:after ?t1)  
(?t4 time-entry:ends ?T2)]
```

```
[meets:(?T1 time-entry:intMeets ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),
```



```
(?t1 rdf:type time-entry:Instant),  
(?t2 rdf:type time-entry:Instant),  
(?t3 rdf:type time-entry:Instant),  
(?t4 rdf:type time-entry:Instant),  
(?T1 time-entry:begins ?t1),  
(?T1 time-entry:ends ?t2),  
(?T2 time-entry:begins ?t3),  
(?T2 time-entry:ends ?t4),  
(?t time-entry:ends ?T1),  
(?t time-entry:begins ?T2)]
```

```
[metby:(?T1 time-entry:intMetBy ?T2) <-  
(?T1 rdf:type time-entry:IntervalEvent),  
(?T2 rdf:type time-entry:IntervalEvent),  
(?t1 rdf:type time-entry:Instant),  
(?t2 rdf:type time-entry:Instant),  
(?t3 rdf:type time-entry:Instant),  
(?t4 rdf:type time-entry:Instant),  
(?T1 time-entry:begins ?t1),  
(?T1 time-entry:ends ?t2),  
(?T2 time-entry:begins ?t3),  
(?T2 time-entry:ends ?t4),  
(?t time-entry:begins ?T2),  
(?t time-entry:ends ?T1)]
```