

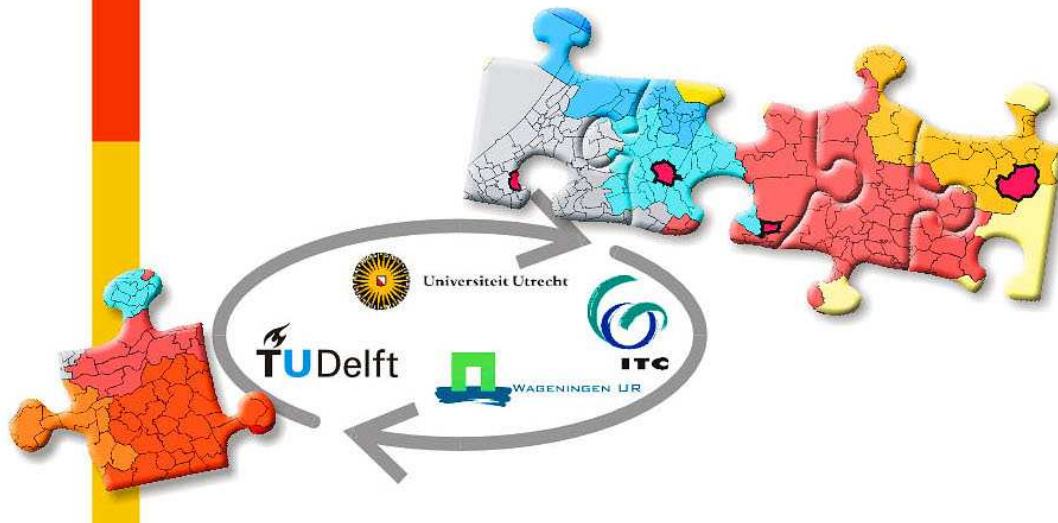
# GIMA

Geographical Information Management and Applications

## Using semantic technologies to design a Spatio-Temporal database

**Master of Science Thesis**  
**Lieke Verhelst June 24th 2009**

Professor: Prof.dr.ir. Peter van Oosterom,  
Delft University of Technology  
Supervisor: Drs. Wilko Quak,  
Delft University of Technology  
Reviewer: Dr. Ir. Arend Ligtenberg  
Wageningen University  
Reviewer: Dr. Willem Robert van Hage,  
Vrije Universiteit Amsterdam





# Abstract

This research deals with the problem of the enormous amounts of data that is streaming from the 'sensor web' into our computer systems. This data is useless to us unless it is properly stored, queried and presented by means of computer systems. These are typically databases, query engines, applications and user interfaces. In particular the manner *how* data is stored in a database determines what information you can retrieve from the system. This is widely known when it comes to traditional relational databases, however when complex data structures such as spatial, temporal and spatio-temporal structures are involved the user of sensor data simply lacks an understanding of this issue. This thesis work addresses this problem by designing a prototype of an expert system that automatically makes the selection of an appropriate technical solution based on information entered by the user of sensor data.

For the design of this prototype, techniques coming from Geo Information science are combined with those from Artificial Intelligence and Computer Science. The Artificial Intelligence techniques used are the building of an ontology and the use of a reasoning system and rule engine. These techniques, called semantic techniques, are typically useful for storing concepts and relations, querying them and drawing conclusions from them. These techniques are compared with one modelling method commonly used in software engineering, namely UML.

The prototype that was built enables the user to enter requirements for the question to be posed upon the sensor data, as well as information about the structure of the data set used. The prototype also contains knowledge of existing technical solutions as well as features of space, time and space-time. A logical component then decides which technical solution is selected based on the information entered by the user. For the logical component two different semantic technologies were possible. One is the use of a reasoner. The other solution is a rule engine. Both solutions were implemented and this resulted in two prototypes, the JessTabDemo and the ReasonerDemo. The capabilities of the two prototypes were evaluated against the predefined prototype requirements. The conclusion was drawn that neither solution satisfied every predefined requirement. For reasons of comparison an imaginary solution based on UML was envisioned. This solution also did not satisfy every predefined requirement. This research concludes therefore that the envisioned instrument can best be built with a combination of UML and semantic technologies. It remains a challenge for the future to combine static solutions, such as databases, with dynamic ones such as ontologies.



# Acknowledgements

A word of thanks goes out to my professor Peter van Oosterom and supervisor Wilko Quak. Thank you also Willem to step in this process at a late time and guide me to expert knowledge of the subject. Thanks Arend for your time and review work. The invited guests Rob Lemmens and Marian de Vries, thank you for making time to give feedback and suggestions.

My home and base Loeki, I owe you much. Your patience, support, love and other help have been essential!

June 2009, Lieke Verhelst

Delft, University of Technology, OTB Institute



# Table of Contents

Abstract.....	iii
Acknowledgements.....	v
Table of Contents.....	1
List of Figures.....	3
List of Tables.....	5
Acronyms.....	7
1 Introduction.....	9
1.1 Research question.....	10
1.2 Methodology.....	11
1.2.1 Scope of the research.....	11
1.2.2 Literature research.....	11
1.2.3 Requirements of the prototype.....	12
1.3 Some remarks on terminology used.....	13
1.4 Thesis outline.....	14
2 Space and Time in databases.....	15
2.1 Temporal semantics.....	16
2.2 Spatial semantics.....	17
2.3 Spatio-temporal semantics.....	17
2.4 Query Capabilities of data models.....	18
2.5 Other database capabilities.....	19
3 Artificial Intelligence.....	21
3.1 Textual notations of knowledge.....	21
3.1.1 Formal logic.....	22
3.1.2 Rules.....	22
3.2 Graphical and textual notations of knowledge.....	22
3.2.1 Object-Attribute-Value Triplets.....	23
3.2.2 Fuzzy Facts.....	23
3.2.3 Semantic networks.....	24
3.3 Semantic web technologies.....	24
3.3.1 Resource Description Framework (RDF).....	25
3.3.2 Web Ontology Language.....	27
3.3.3 Ontologies.....	27
3.3.4 Working with ontologies.....	28
4 Software engineering.....	31
4.1 Model Driven Architecture.....	31
4.2 Unified Modelling Language.....	32
4.3 Ontology Definition Metamodel.....	34
4.4 UML versus OWL.....	34
5 Building the Prototype.....	36
5.1 Protégé.....	38
5.2 Jess.....	45

5.3	JessTab .....	46
5.4	The reasoners: Pellet and RACER .....	48
6	Building the ontology .....	51
6.1	The base ontology .....	55
6.2	Refining the ontology .....	60
7	Jess rules .....	64
7.1	Property multivalues .....	70
7.2	OWL data structure .....	72
8	Using the reasoner .....	76
9	Comparing the solutions .....	81
10	Conclusions and recommendations .....	84
10.1	Summary and discussion .....	84
10.2	Answers to sub-questions .....	85
10.3	Main conclusions and recommendations .....	87
10.4	Suggestions for further research .....	88
	Bibliography .....	89
	Appendix 1 Example Ontologies .....	93
	Appendix 2 JessTab .....	95
	Appendix 3 The OWL Ontology .....	99

# List of Figures

Figure 1: An object–attribute–value (O–A–V) triplet .....	23
Figure 2: An O-A-V triplet with a certainty factor.....	23
Figure 3: A membership function of the age classification against the age number .....	23
Figure 4: A simple semantic network .....	24
Figure 5: The semantic web layered-cake, taken from [12] .....	25
Figure 6: An example of a RDF graph, taken from [13] .....	26
Figure 7: A web of ontologies connected to applications, taken from [10] .....	28
Figure 8: Example of a class diagram .....	33
Figure 9: An example of a use case diagram, taken from [24] .....	33
Figure 10: Protégé ontology browser.....	39
Figure 11: Protégé Class editor .....	40
Figure 12: The Protégé Property Editor.....	41
Figure 13: The Protégé Individual editor .....	42
Figure 14: The Protégé Form editor .....	43
Figure 15: The user interface for entering information about the data set.....	44
Figure 16: The JessTab interface in Protégé .....	47
Figure 17: The knowledge of data models stored in the ontology .....	55
Figure 18: An ontology of time created by NASA, taken from [33].....	56
Figure 19: An ontology of time created by W3C, taken from [34].....	56
Figure 20: The basic classes and properties of the ontology.....	57
Figure 21: The basic relations between the model entities .....	58
Figure 22: The Feature class with subclasses.....	59
Figure 23: The Feature class with all the subclasses and sub-subclasses.....	59
Figure 24: The Solution class with subclasses .....	60
Figure 25: All classes and sub-sub-classes with properties.....	60
Figure 26: A graphical representation of Example 1 .....	62
Figure 27: A graphical representation of Example 2 .....	62
Figure 28: A graphical representation of Example 3 .....	63
Figure 29: The result of the JessTabDemo as the YourSolution instance.....	65
Figure 30: The result of the JessTab demo in the JessTab console .....	65
Figure 31: The UserQuestion individual converted to a Fact.....	67
Figure 32: Multiple values stored in the property supportsSQueryCapability .....	71
Figure 33: The data structure of the facts .....	73
Figure 34: An inferred individual .....	77
Figure 35: The rules entered in the Protégé editor (Example 2).....	78
Figure 36: The result of inferring individuals with a reasoner .....	79



# List of Tables

Table 1: The knowledge of the data model capabilities stored in the ontology .....	52
Table 2: Properties of DBMS included in the ontology .....	54
Table 3: Mapping the contents of the knowledge table .....	61
Table 4: The reasoner solution versus the JessTab solution .....	81



# Acronyms

AI	Artificial Intelligence
CL	Common Logic
CLIPS	C Language Integrated Production System
CIM	Computational Independent Model
CWA	Closed World Assumption
DBMS	Database Management System
LHS	Left Hand Side (used in Jess coding)
MDA	Model Driven Architecture
NFR	Non-Functional Requirements
ODM	Ontology Definition Metamodel
OGC	Open Geospatial Consortium
OMG	Object Management Group
OO	Object Oriented
OWA	Open World Assumption
OWL	Web Ontology Language
PIM	Platform Independent Model
PSM	Platform Specific Model
RDF	Resource Description Framework
RHS	Right Hand Side (used in Jess coding)
SRS	System Requirement Specification
ST	Spatio-Temporal
SWE	Sensor Web Enablement
UML	Unified Modelling Language
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium



# 1 Introduction

Since the vast expansion of wireless technologies it has become much simpler and cheaper to use sensors for the collection of geo data. Because of both the increasing area coverage of wireless antennas and global positioning systems as well as improved power supply solutions, sensors can nowadays be placed practically everywhere. Researchers of various academic disciplines understand the opportunities that sensors offer: when sensors are attached to the Internet as a sensor network, a real-time sensing system of systems is accomplished.

The Open Geospatial Consortium (OGC) has embraced this idea of a 'sensor web'. This resulted in the specification of a suite of standards called Sensor Web Enablement (SWE). This opens up numerous possibilities for research in areas of environmental monitoring, transportation management, public safety, disaster management, and many other domains [1].

However promising this may seem, 'data' is not the same as 'information', and is it not the latter we are interested in. The next challenge therefore is to manage the large amounts of data that are streaming into our computer systems in order to store them, analyse them and present them as information in the format that corresponds to a user's needs.

Geo sensors can help us to understand how the earth responds to climate change. Measurement of CO<sub>2</sub> values, sea surface temperature, iceberg locations and the like give us information of the current climate situation. When this information is processed appropriately we can use it in advanced models that not only monitor current measurements but also estimate future values.

This thesis work is based on the premise that users of sensor data are often ignorant of these opportunities. Users neglect the fact that sensor data essentially contains (apart from the measurement value) both a spatial and a temporal component that should be managed in a spatio-temporal data context. Because this is not understood, a large part of potential information coming from sensors is not disclosed.

The *objective* of this work is therefore to provide users of sensor data with an instrument that advises them how to store the data in a database system in such a way that it optimally fits their needs. When this is accomplished the research result will hopefully contribute to a more effective use of sensor data as an instrument to understand global changes.

To address this objective it is necessary to understand the characteristics of spatio-temporal data and how they are implemented in technical solutions. In addition to that it is essential to present a basic understanding of what users of sensor data would like to learn from this data, in other words what kind of data questions they would pose.

Most likely the instrument will present more than one possible solution. A weighing mechanism to compare the proposed solutions can position them against each other.

## 1.1 Research question

Since this research is executed in the context of geo-information science, the 'instrument' that is mentioned in the objective will be created as a software system. Several methods to achieve this are available as specifications from computer industry consortia.

The main research question is:

*How to design and create an instrument that uses the concepts of sensor data (e.g. sample frequency, size of data file, structure of sensor data) and the sensor data user requirements in order to design a logical and physical database model.*

The context of this question will be investigated by answering a series of research sub-questions regarding sensor data, sensor data user requirements and matching technical solutions. The questions are:

- what are the most important characteristics of spatio-temporal data?
- what are user questions and how can they be categorised?
- which technical solutions are available that take care of a proper handling of spatio-temporal data?
- what widely supported specification(s) can we use to create the instrument?
- which tools can be used to design and create the instrument?
- how can we model the relation between sensor data concepts and a suitable database model?
- how can we assign a weight qualification to compare proposed instrument outcomes?

## 1.2 Methodology

To answer the research questions a literature study was performed and after this a prototype was designed.

### 1.2.1 Scope of the research

This research touches many research areas and therefore it is necessary to set some bounds. Describing concepts of time and space is in itself a task which, by the metaphysical complexity of it, can never be completed. Also, there are technical (database) solutions available in large numbers and many are still in development. Therefore the most important concepts and technologies are targeted.

Many methods and languages of software engineering are available. In this research only the ones from leading consortia like the World Wide Web Consortium (W3C) or the Object Management Group (OMG) are examined for design of the instrument. Other standards from organisations such as OGC and the International Organisation for Standardisation (ISO) are not used to design the instrument but must be referenced to when definitions of geographical and other concepts are necessary to the reader.

The timeframe and objective of the MSc thesis only allows the creation of a prototype in which principles and methods are tested.

### 1.2.2 Literature research

The scope of the literature research was narrowed down by defining main concepts and keywords. These keywords were: spatio-temporal, (database) modelling, semantics, system requirements, ontology.

Thanks to the Internet and digital university libraries many, many academic papers and books were found. This made it possible that the research that started with a blank sheet could be completed with a clear result. The use of Wikipedia<sup>1</sup> must be mentioned here. This greatly helped to understand the basics of concepts described in academic papers that sometimes only touched a fraction of the whole idea. Because the authors of Wikipedia are not (always) identifiable this has never been used as a source of information to cite from or refer to directly.

---

<sup>1</sup> [www.wikipedia.org](http://www.wikipedia.org)

### 1.2.3 Requirements of the prototype

The idea behind the prototype was to build an instrument that helps the user to design the most optimal logical and physical database implementation that meets the requirement of the user. It is not possible to catch *all* the separate possibilities that influence this design. However, the prototype must support some basic concepts. First, the user must be able to *enter information* about the data set he wants to work with and what application the database must be able to support. Secondly, the prototype must be able to *store existing knowledge* of spatio-temporal database design, and be *flexible enough* to adapt new knowledge. The prototype must be able to *automatically guide* the user to the best solution to his or her needs, and if there is more than one way to do it, the system must provide a *ranking order* for all the presented solutions. When possible, the system must *provide code* for automatically generating the logical and physical database design.

The system is typically intended for the following context. A user has collected data with a sensor. The data is available to the user in a file, organised in a certain format. Both the time and the location information are available in this file. The file can be static (it is retrieved and disconnected from the sensor) or constantly growing (the sensor is streaming data into the file).

If the sensor produces a file in which time or space is not included as data, this information must be retrieved from another source (most likely another table) via a lookup function.

The data file would thus have the following information, parted by a separating character like colon, tab or space:

- sensor-id
- measured value(s), this can be a complex value
- the location (in lat, lon, h or x , y, z)
- the time

The user then wants to store the data in a database, because this provides better use and maintenance functionality than a file. Role based security, performance optimisation, concurrent use of data, data integrity, and controlled recovery from hardware failures are typically reasons for giving preference to a database.

This data must be analysed by means of asking questions about the data, and the results need to be displayed. For this purpose a custom application will be built. The user is not aware of the fact that how the data is stored in the database eventually determines the complexity of the application code. Some type of application requirement analysis and application modelling would therefore be helpful for this user to determine how the database must be designed.

To indicate how many application types are possible some examples are outlined here.

In the most basic situation the user is primarily interested in the attribute values of the data set. The user does not need to perform spatial calculations like measuring distance or calculating a buffer. Also more complex computations like interpolating the values between measurement points are not required. The user is also not interested in comparing the values in a complex time context, such as comparing data stored in different time structures.

In situations where the user wants to work with the spatial characteristics of the data techniques that support this become necessary. Examples are interpolating the measurement values and displaying the result as a spatial raster and relating spatial objects to each other (determining objects that are *inside*, *touching*, *overlapping* etcetera). Spatial databases are optimised for these kinds of applications. They provide built-in operations and functions that perform calculations that otherwise had to be programmed in a common database application. In addition to that, spatial databases provide optimised data access methods that enable fast spatial data retrieval.

When the user wishes to compare data values in complex time situations it might be more efficient to use a temporal database than a relational database. A temporal database is optimised to store, retrieve and work with temporal data. Examples are situations where it is important to record changes and keep the old values instead of overwriting them. Application areas are amongst others monitoring statistics, weather and seismic readings [2].

In situations where the user wants to identify the change of a spatial object over time a spatio-temporal database is necessary. This kind of database supports the complexity of time in relation with space, such as movement of objects or change in shape.

The outcome of the prototype building is described in chapters 5, 6, 7 and 8.

### 1.3 Some remarks on terminology used

The reader will notice that throughout this document more than one terminology is used for the name of the instrument to be built. Sometimes it is called 'instrument', later on 'system', 'model' and 'prototype'. They all refer to the same thing. The name 'instrument' is used when the technical implementation is not known or not important in the context. 'System' is used when its character was revealed to be technical. The word 'model' is used in the phase of 'modelling' and finally in the implementation stage the word 'prototype' is used.

Also, some explanation about 'modelling' and 'engineering'. In science, models are created to simulate and study real world phenomena. By creating a model we can work at a higher level of abstraction. This is achieved by hiding or masking details, by showing the big picture, or by focusing on different aspects of a prototype.

Models help us understand how things work and on the other hand we use them to make predictions.

In software development 'modelling' is the phase in the designing of software applications before final coding. A model helps to ascertain that business functionality is complete and correct, that end-user needs are met and that requirements for scalability, robustness, security, extendibility, and other characteristics are there before any line of code is written [3].

The more general term 'engineering' is used to describe the process of designing and constructing technical systems<sup>2</sup>.

## 1.4 Thesis outline

The thesis document is outlined according to the following structure. First the results of the literature research are documented. Since the literature topics are substantial and diverse they are divided between three chapters, namely 2, 3 and 4. Chapter 2 takes care of the Geo Information element of this research: space and time in databases. Chapter 3 covers the topics coming from the science of Artificial Intelligence that were used in the research. Chapter 4 contains a background description of methods of software engineering. These three chapters describe which part of the sciences of Geo Information, Artificial Intelligence and Computer Science were used and combined to produce the research result.

Chapters 5, 6, 7 and 8 describe how the research product was created. Chapter 5 describes which tools were used and why, chapter 6 describes how the product was built and the chapters 7 and 8 describe two ways of how the product can be used for answering the research questions. Chapter 9 compares these two ways.

Chapter 10 contains conclusions and answers the research questions. Recommendations for further research take care of open ends that the research could not cover.

---

<sup>2</sup> Adapted from Cobuild advanced dictionary

## 2 Space and Time in databases

This chapter describes how the concepts of time and space in information systems play a role in answering the research question.

In computer systems, a data model defines how the data is stored in the database management system (DBMS) and how it can be retrieved. A data model consists of a set of definitions about data types, relations and operations. As a result, the data model determines the capabilities of the database system.

In most cases it is sufficient to use a common relational data model. For some phenomena of the real world however we need a special data model. This is valid for the phenomena of time and space. Using a special data model enables us to effectively work with the data.

A **spatial data model** is optimised to work with spatial data. It has spatial data types such as raster and point, line and polygon. The data model is supported by dedicated spatial access methods (called indexes) and functions. Examples of DBMS that support a spatial data model are Oracle Spatial and PostGIS. They are both extensions of a relational database (Oracle and PostgreSQL respectively) that implement the spatial data model as a special data type object next to the normal data types as text, date and number.

A **temporal data model** is used in applications where time is the most important element. An example of a DBMS supporting this data model is Informix TimeSeries DataBlade (an extension of Informix Dynamic Server) [4]. This type of database is optimised to store and retrieve time related data. A typical data type for this DBMS is the TimeSeries data type. A TimeSeries can be either regular (at defined time intervals) or irregular. In a common database time is usually stored as a column value in a record, just like the other values in the record. The TimeSeries data type does not store the date and time with the other data in a record. The data is stored as an offset to a known beginning date (origin). Each TimeSeries can have a different origin. [2]. This concept reduces storage space, improves performance and makes data querying easier.

To model the relation between time and space we need the **spatio-temporal data model**. Currently there are no commonly used DBMS that have a spatio-temporal model, however some immature versions are listed in [5].

The literature study of this thesis work was executed on [6], [7], [8] and [5], and [9]. The book by Gianotti and Pedreschi (eds.) ([6]) covers a wide area of topics, ranging from data modelling and data mining to privacy aspects of spatio-temporal data. This book was used as introduction to the vast area of spatio-temporal research. It was also used for setting the research scope.

The works by Pelekis [8] and Pelekis, Theodoulidis, Kopanakis and Theodoridis [5] contain a historic overview of the academic work on spatio-temporal modelling over the last two decades. The book by Güting and Schneider [7] concentrates mainly on continuously changing objects in contrast to discretely changing objects. This book was used to complement the other works with regard to the information of data models of continuously changing objects.

The article by Lemmen [9] was used as a reference for existing spatial database solutions and their capabilities.

It is the objective of the thesis to advise the user of sensor data with a solution for their intended application. As will be outlined in the thesis, the data model is the most important part of this solution. The works of Pelekis cum suis describe and compare many spatio-temporal models. Therefore it is chosen as the main reference for this research.

In his work [8] Pelekis documents characteristics of space, time and space-time. He explains how they are (or are not) supported by spatio-temporal data models that have been developed since the early nineties until the early 2000s. Because this information is used frequently in the thesis work this is summarised in the following paragraphs: all information in paragraphs 2.1, 2.2, and 2.3 is taken from [8].

In [5] it is emphasized that what makes a database system fit for purpose is both how semantics of space and time are incorporated in a data model and the capacity of the database to support a type of query. In the design of the system this has been taken as the main criterion for the determination of the advice to the sensor data user.

## 2.1 Temporal semantics

Listed in [8] are the temporal semantics: granularity, density, reference, modelling of time stamp, representation of time, time type, order, span and lifespan.

**Granularity** can be described as the length of a partition of time on the time axis.

The value of **density** is either discrete (isomorphic to integers) or continuous (isomorphic to real numbers).

These two concepts are related to the sample frequency of the recording sensor.

**Time reference** is a criterion that describes whether time is considered absolute (exact points on the time axis) or relative (two weeks before..).

Different models use different methods for **modelling the timestamp**. Examples are: 'mm-dd-yyyy' or 'Monday August 12 2001'.

The **representation of time** can be assigned to different levels in the model: to different parts of the geographic object (polygon/line or vertex level), to the temporal event or to a combination of the two.

**Transaction time** refers to the moment the item is recorded in the database, while **valid time** refers to the time that the event occurred in the real world.

**Time order** refers to the way to describe the perspective of time, that of time as an arrow, as a reoccurring event (circle) or other ways.

**Span** refers to whether the duration of an event is supported.

**Lifespan** refers to the fact whether the model keeps track of history.

These concepts become typically important while building the application.

## 2.2 Spatial semantics

The spatial semantics listed in [8] are structure of space, orientation/direction, measurement, operations and topological relationships.

The description **structure of space** is used to refer to the two basic approaches of storing geographic data: raster and vector.

**Orientation/direction** describes if characteristics like 'on the left side of, to the right' are supported.

**Measurement** refers to the possibility of obtaining measurement values like distance, perimeter, length.

**Operations** describes whether operations like 'equal, bigger, smaller' are supported.

The eight **topological relationships** are disjoint, meet, overlap, equal, covers, covered by, inside, contains.

## 2.3 Spatio-temporal semantics

The spatio-temporal semantics ([8]) are: data types, primitive notions, change, object identities, continuous change, discrete change, movement, functions, evolution, measurement/topology and dimensionality.

**Data types** refers to which data types (such as point, line, polygon for space, interval and instant for time) are supported by a model.

**Primitive notions** refers to how the developer of the data model has created an abstraction of the real world.

The concept **change** has been used to compare how the models deal with changes in time and shape/size.

Change can be either **continuous** or **discrete**.

**Object identities** refers to the notion that objects might be affected by change so much (for example splitting, unifying) that they must also change their identity.

**Movement** refers to the criterion if a model supports change of position and/or shape in time.

**Functions** indicate if there are defined functions like 'creation, evolution' available in the model.

**Evolution** is used to compare models on the availability of functions able to calculate for example velocity and acceleration.

The **measurement/topology** criterion lists whether models support geometrical measurements and topological relationships.

**Dimensionality** relates to how models support dimensions. The 2<sup>nd</sup> and 3<sup>rd</sup> dimension being usually adapted by GI systems, some spatio-temporal models can support higher dimensions like the fourth (x, y, z, t) or even fifth when past and future are modelled in a separate time dimension.

Instances of spatio-temporal objects have spatial, temporal and spatio-temporal characteristics like the ones described in the previous paragraphs. It is the *data model* that determines whether these characteristics can be interpreted by the database system or not.

One other very important capability of the data model determines its usability: the *query capability*.

## 2.4 Query Capabilities of data models

The query capability is the capability of the data model to answer questions that the user (by means of a query) poses on the data.

In [8] the query capabilities of the data models have been briefly listed. A more detailed description of what is meant is recorded in [5]. Unfortunately the definitions of the query capabilities given in these works are not very precise. They allow for more than interpretation for what is meant. Since we want to use this information in our system an assumption must be made.

It is assumed that the *name* of the query capability identification given in [8] indicates the capability. As a consequence it is assumed that:

- 'Simple' means that only one object of space or time is involved in the query.
- 'Relation' means that the query is capable of relating more than one (space or time) object to the other.
- 'Range' means that in the time dimension the range attribute is used.
- 'Behaviour' means that the relation between time and space is evaluated.

As such the nine possible query capabilities are:

**Attribute queries** – this type does not query the space or time element of the object, only the attributes. An example is: *'what is the identifier of object o'*

**Simple spatial queries** – are queries concerning one or more spatial objects, however the relation between spatial objects is not queried. Example: *where is the object with attribute value x located?*

**Spatial relationship queries** – these query the spatial relations between objects. An example is: *'which objects are inside area a'*

**Simple temporal queries** – refer to a situation at a specific point in time. An example is: *'what is the attribute value of an object at time t'*

**Range temporal queries** – refer to temporal ranges or periods: *'how does an object attribute value change over a given period'*

**Temporal relationship queries** – are queries that refer to a relation of two temporal entities. Example: *'find attribute values of objects that relate in a time value t1 and a time value t2'*

**Simple spatio-temporal queries** – are referring to the spatial condition in a temporal instant (for example *'what is the shape of an object at time t'*).

**Range spatio-temporal queries** – are referring to the spatial condition over a temporal range (for example *'what happens to a spatial object over a given period'*)

**Behaviour spatio-temporal queries** – are referring to changes in time and space simultaneously and continuously such as speed, velocity and change rate. Example: *'when/where did the spatial object reach its maximal rate of spread'*

## 2.5 Other database capabilities

One important element of spatio-temporal data that is a bit under exposed in [8] and [5] and therefore in the previous paragraphs is the effect of the sample frequency of the sensor. The sample frequency of the sensor determines eventually how large the data file will become and how fast it will grow. This is extremely important for the performance of the database storage and retrieve (query) capabilities.

Solutions for performance in a database are access methods (indexes) and clustering. In [9] the various DBMS are functionally compared on these capabilities. It is very difficult however to indicate which capability has the best performance result for which situation. This would require a benchmarking comparison and this is technically a very difficult and questionable exercise. This is why these database capabilities have only partly been included in the prototype. They do not take part in the calculation of the solution, but they are included in the instrument as knowledge of existing database solutions.

This chapter has introduced the complexity of time and space modelling in computer systems. This is one of the three topics that were used for answering the research question. The others are the topic of Artificial Intelligence and Software engineering. The

concepts of Artificial Intelligence that were used to produce the research product are the subject of the next chapter.

## 3 Artificial Intelligence

This chapter will introduce the necessary words, concepts and techniques coming from the science of Artificial Intelligence that were used to make the research product.

In [10] Artificial Intelligence (AI) is described as the branch of Computer Science that studies the nature of human knowledge. Its objectives are to understand the concept of knowledge and to develop methods to simulate intelligence. Important study areas of AI are knowledge storage and retrieval, knowledge acquisition, knowledge representation and reasoning.

The AI study area of knowledge storage examines how knowledge can best be encoded in a suitable format before it can be stored into computer memory. To retrieve the knowledge, one has to follow the inverse process.

Reasoning stands for using the stored knowledge together with problem-solving strategies in order to find new information such as conclusions and explanations.

Every piece of knowledge has a certain degree of certainty. It is often incomplete and imprecise and this causes problems while reasoning. A solution for this has been found in making assumptions about the scope of the knowledge. The open world assumption (OWA) refers to the idea that we cannot say that something doesn't exist until its non-existence is explicitly stated. As a consequence, if something hasn't been stated to be true, it cannot be assumed to be false — it is only assumed that 'the knowledge just hasn't been added to the knowledge base'.

The opposite of the open world assumption is the closed world assumption (CWA). When something is not known to be true in CWA it is defined as false.

It is an important assumption in AI that a human mind has mental representations analogous to computer data structures. It is also assumed that the reasoning procedures of the mind are similar to computational algorithms.

The knowledge representation methods used in AI are textual or a combination of graphs and text. Some are described in [10] and they listed in the following sections.

### 3.1 Textual notations of knowledge

The textual notations are notations that describe knowledge, such as formal logic and notations that work with knowledge, for example rules.

### 3.1.1 Formal logic

The most common representations of knowledge are first-order predicate calculus and description logic. These have precisely defined grammars that can therefore be read and interpreted by both humans and computers.

First order predicate calculus is a type of predicate calculus. 'Predicate calculus works with objects (terms), properties (unary predicates on terms), relations (n-ary predicates on terms) and functions (mappings from terms to other terms). It is called first-order because it allows quantifiers to range over objects (terms) but not properties, relations, or functions applied to those objects' (taken from [11]). An example of a first order predicate logic statement is (adapted from [10]):

```
in(boy, room)  $\cap$  in(mother, garden)  $\rightarrow \neg$  see(mother, boy)
```

The above statement represents the following: 'the boy is in the room and the mother is in the garden, therefore the mother cannot see the boy'.

Description Logic (DL) is a language that contains a set of constructs for describing concepts and their relationships. It is described in more detail in paragraph 3.3.4.

### 3.1.2 Rules

A rule creates a structure that relates one or more conditions to one or more conclusions or actions. Usually this is done via an IF ..THEN.. construct.

For example:

```
IF The door is locked  
AND I have a key  
THEN I can open the door
```

The statement after the IF is usually a fact, the statement after the THEN is usually an action or the creation of a new fact.

Uncertainty can be included in facts by using a *certainty factor* in the expression:

```
IF The door is a bit closed  
AND I have some sort of a key  
THEN I can probably open the door
```

More about rules in paragraph 3.3.4.

## 3.2 Graphical and textual notations of knowledge

Images and graphs are a powerful means of representing knowledge because the human mind is used to interpreting images. For computers it is more difficult to work with

images, therefore in AI graphical representations usually have a textual counterpart. The figures in the following paragraphs are taken from [10].

### 3.2.1 Object-Attribute-Value Triplets

Object-attribute-value (O-A-V) triplets are used to represent facts about objects and their attributes. In the image below the oval represents an object, the arrow the attribute and the box the attribute values. Objects usually have multiple attributes, therefore there may be more than one arrow-box combination for each oval.

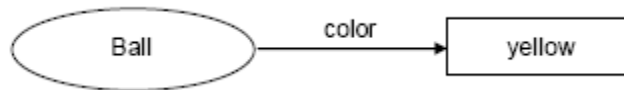


Figure 1: An object-attribute-value (O-A-V) triplet

The textual counterpart of this kind of triplets is described in paragraph 3.3.1.

### 3.2.2 Fuzzy Facts

Just like in textual knowledge notations, graphical notations can have certainty factors. This is usually illustrated by adding an extra box with a value, a numeric representation between 0 and 1 of the certainty: 0 being 100% uncertain and 1 being 100% certain.

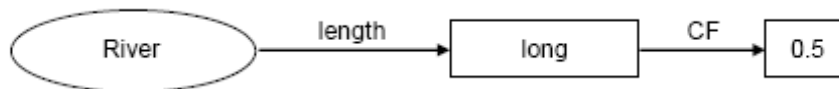


Figure 2: An O-A-V triplet with a certainty factor

The distribution of the certainty of a fact can be outlined in a graph representing the certainty factor as a function of the fact. This graph is called a membership function.

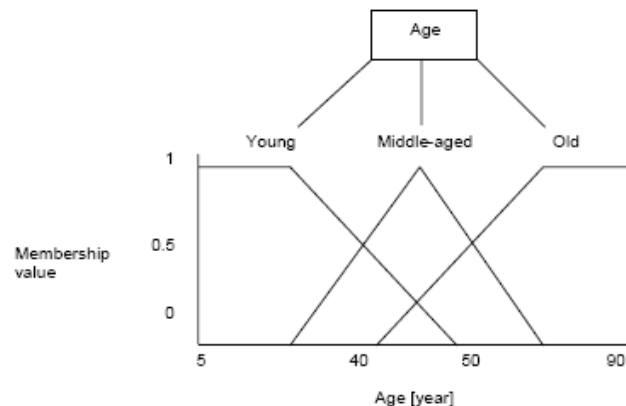


Figure 3: A membership function of the age classification against the age number

### 3.2.3 Semantic networks

In the late 1970s and early 1980s the first semantic networks were developed. Just like O-A-V triplets semantic networks are based on a graphical representation of the relations between objects or concepts. However in semantic networks there is more freedom to describe the relation and the concept. For example, a relation can be class-subclass or class-instance or class-property. This makes it possible to create a series of triplets that eventually make up a network, as can be seen in Figure 4 .



**Figure 4: A simple semantic network**

At that time the semantics of networks were not clearly defined. As a consequence two systems could be created using the same name structure but behaving very differently. This problem was addressed the late 1980s (see paragraph 3.3.4).

A semantic network that describes a particular concept domain is an ontology. This is described in paragraph 3.3.3.

## 3.3 Semantic web technologies

Tim Berners-Lee, the inventor of the World Wide Web, envisioned in the year 2000 the future of the Internet. In this vision he included knowledge management techniques used in AI to create a more intelligent network. He called it 'the Semantic Web' and pictured it as a layered-cake of technology. The higher levels in the cake build upon the lower levels. The language syntax used in all the layers is based on XML.

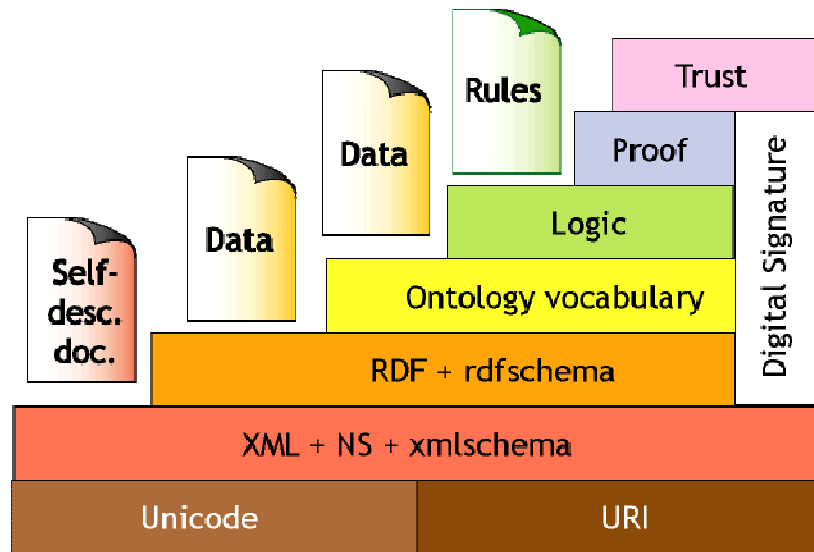


Figure 5: The semantic web layered-cake, taken from [12]

The lower layers represent low level information like the identifiers and the meta environment (the namespace and XML language). Climbing up the layers the information changes to a richer format via two textual representations of knowledge – RDF and OWL - to rules, proof and eventually trust.

RDF and OWL are XML based notations that are suitable in distributed environments like the Internet. They are both World Wide Web Consortium (W3C) specifications.

### 3.3.1 Resource Description Framework (RDF)

The Resource Description Framework (RDF) was developed to represent information about resources on the Internet [13]. Although originally designed for the web, it can be used as a common framework to exchange information between different applications, even to those for which it was not originally created. It is based on the O-A-V triplet concept. An RDF triplet contains an object, a predicate and a subject.

The description can be notated in the forms of Uniform Resource Identifier (URI)'s in a XML like format. An example is [13]:

```
<http://www.example.org/index.html> (the object)
<http://purl.org/dc/elements/1.1/creator> (the predicate)
<http://www.example.org/staffid/85740> (the subject)
```

The triplets can also be displayed in a graph. Figure 6 represents the graph of the triplet above.

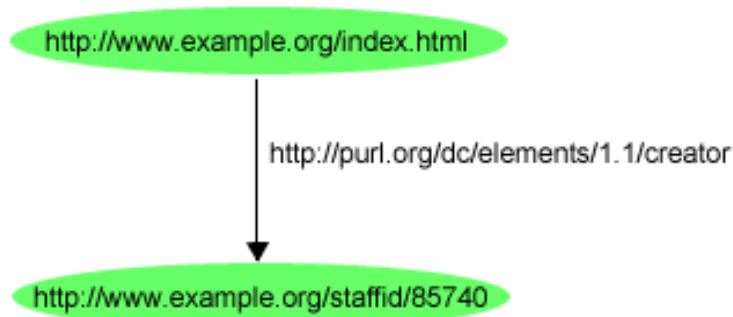


Figure 6: An example of a RDF graph, taken from [13]

This example contains the following information: the web document 'www.example.org/index.html' has the property 'created' that is linked to 'something' that has the staff id 85740. This might at first seem a bit odd however it is based on the idea of how *data* is linked in the open world. This is described in [14]. The principle of linking data is the same as with HTML: use URI's as names for things and use the HTTP protocol to transport data lookup requests. As soon as we have retrieved the linked data this will become more understandable and lead to the information that the web document 'www.example.org/index.html' has been created by a person with staff id 85740. Linking the staff id with people's names in a table will eventually lead to something like 'the web document [www.example.org/index.html](http://www.example.org/index.html) is created by John Smith'.

RDF schema (RDFS) is the implementation of RDF. With RDFS you can build connecting triplets that describe phenomena (as classes) and their properties. We use the words "classes" and "properties" just like in Object Oriented (OO) programming, however RDF differs from OO in this. In OO a class is defined in terms of the properties of its instances while RDF schema defines properties in terms of the classes of resource to which they apply. As such in RDF schema the approach is property-centric: the emphasis is on the property and not on the class or instance. One benefit of this property-centric approach is that can be used to express characteristics (properties) of phenomena.

There is a limitation to the use of RDFS. It is limited to simple descriptions. RDFS cannot describe types of relations between classes (such as 'disjoint'), it cannot handle cardinality, equality or other characteristics of properties. This is available in a richer version of RDF, namely Web Ontology Language (OWL).

### 3.3.2 Web Ontology Language

The recommendation for OWL, which consists of six documents<sup>3</sup>, was published in 2004 by the Web Ontology Working Group as part of the W3C Semantic Web Activity. The idea behind the development of OWL is that it is designed for applications that need to process the content of information rather than just presenting information. OWL has a greater machine interpretability of Web content than XML, RDF and RDF Schema because it provides a large vocabulary along with a formal semantics [15].

There are three OWL sublanguages and each has its specific purpose [16]:

**OWL Lite** is useful for users that primarily need a classification hierarchy with simple constraints.

**OWL DL** (DL stands for description logic). This version of OWL is for those users who want to use reasoning capabilities

**OWL FULL** is the version that is the most expressive and flexible to use. It is used to capture and describe knowledge when the syntax of OWL-DL and OWL-Lite falls short. OWL-Full is too rich in syntax to be used for reasoning.

In general OWL is used to explicitly represent the meaning of concepts and their relationships. A representation of concepts and relations of a specific domain is called an ontology.

### 3.3.3 Ontologies

The fourth layer of Tim Berners-Lee's cake (Figure 5) is the ontology layer. Ontologies can be defined as connected webs of concepts and relations that contain knowledge (Figure 7). They are encoded in RDF or OWL and therefore machine-understandable and machine-processable. In principle they are a basis for web based application knowledge processing and knowledge sharing.

---

<sup>3</sup> These are OWL Overview, the OWL Guide, the OWL Reference, the OWL Semantics and Abstract Syntax document, the OWL Web Ontology Language Test Cases document, the OWL Use Cases and Requirements document. They reference to each other. In this thesis the OWL Guide and OWL Overview are used. They are referenced to in the text where appropriate.

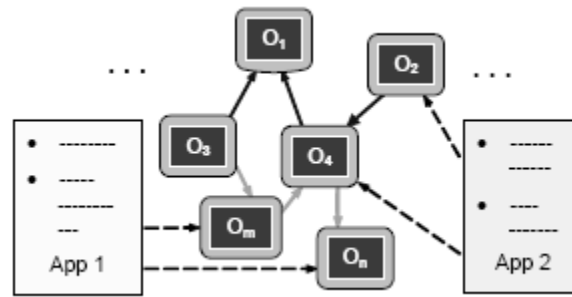


Figure 7: A web of ontologies connected to applications, taken from [10]

Since the development of this concept there has been a community who adapted these ideas. Ontology engineering has become popular amongst scientists to document their theories, ideas and concepts. Until now thousands of ontologies were created. They are accessible by machines and humans on the Internet. This opens up possibilities for connecting these ontologies and to peer-to-peer ontology development.

Some available ontologies related to geo-information science are listed in appendix 1.

On the Internet semantic web technologies are used to improve the interoperability of services. Because ontologies formalise concepts and relations they provide a means for improvement of discovering web services. How this works within the specific context of geo web services has been described in [17].

### 3.3.4 Working with ontologies

The information stored in ontologies can be disclosed using knowledge retrieval technologies. Of these technologies a reasoner and a rule engine are described here. A reasoner uses Description Logic syntax while a rule engine uses programming language.

#### Description Logic

Ironically enough, the early semantic networks suffered from a lack of semantic consistency. One of the first systems that addressed this issue was KL-ONE in 1985. KL-ONE introduced most of the key notions that were the basis for Description Logics [18]. Description Logic is a concept language that contains a set of constructs for describing concepts (in OWL: classes) and their relationships (in OWL: properties). An example of a class construct is the concept conjunction statement  $A \cap B$ . This statement describes the set of individuals that belong to both class A and B. Similarly other constructs exist like the concept disjunction ( $\cup$ , 'or') and concept negation ( $\neg$ , 'not').

Important property relationships are the so called *value restrictions*: the existential restriction  $\exists$  and the universal restriction  $\forall$ .

$\exists p.A$  describes the existence of at least one relationship along property  $p$  to an individual of class  $A$

$\forall p.A$  implicates that all the individuals that are in the relationship  $p$  with the concept being described belong to class  $A$ .

Description Logic is based on the open world assumption (OWA).

### Reasoners

A reasoner is a piece of software that uses DL to run against ontologies and provide services. Some of these services are the checking of [19] [20] concept satisfiability, subsumption, consistency and instance checking. In OWL classes are created with descriptions that specify the conditions that must be *satisfied* by an individual for it to be a member of the class. By using the DL syntax a reasoner can verify if the ontology is *consistent*. A class is inconsistent when it is not possible for it to have any instances. This is for example the case when contradictory properties are assigned to the class.

Another service is *subsumption* testing. This refers to the ability to determine whether a super-class/subclass relationship exists.

*Instance checking* can be used to find out if an individual is a member of a class, even though it is not explicitly assigned to a class in the ontology. The reasoner can deduct this information from the class relation descriptions.

Using this service so called *inferred individuals* can be identified. An example:  
If we have defined the relation between Parent and Child as:

```
Child hasParent Parent
```

and its inverse property as

```
Parent hasChild Child
```

and we have created an individual John from class Child and an individual Mary from Parent then

```
Mary hasChild John
```

can be inferred to:

```
John hasParent Mary
```

## Rule engines

The most common method of programming is procedural programming [21]. In procedural programming the code is instructive and ordered. Therefore it is very suitable for problems in which the input is known and the sequence of steps is clear. Rule-based programming is declarative, meaning that it declares instructions to be executed whenever a condition is true. This kind of programming is useful in situations where the input is fragmented or in other situations where exact algorithmic directions to solutions are not present. Requesting information from an ontology is such a situation because the information stored in the ontology is not known to the program code (rules) before it runs.

If we attempted to search an ontology with a procedural program we would need to write many if-then-else statements and pre-think a solution for every situation where the information is not available or not complete.

In a rule-based program only rules are written, the *rule engine* determines when these rules are to be applied.

The combination of a knowledge base and a rule engine is often called an *expert system*.

In contrast with Description Logics, most rule engines are based on the closed world assumption (CWA).

This chapter has introduced the main technologies and concepts of the science of Artificial Intelligence that were necessary to build the research product. These technologies were combined with those of Geo Information science (as outlined in chapter 2) and of Computer Science. The technologies and concepts from Computer Science that were used are Software Engineering. This is the subject of chapter 4.

# 4 Software engineering

This chapter will outline which methods and technologies from Computer Science can be used to produce the research product.

Software systems are designed in close cooperation with the users of the system. The total process of gathering and specifying requirements for the system is called System Requirements Specification (SRS). A large part of the requirements contributes to the working functionality of the system; however a substantial part is non-functional. These are referred to as Non Functional Requirements (NFR). Examples are performance, security, user community support and cost.

A software system is engineered using methods, software languages and tools. When the use of this is standardised, this contributes to better system interoperability and reuse of code.

The Object Management Group (OMG)<sup>4</sup>, founded in 1989, is an open international consortium of institutions in the computer industry that pursues to develop methods and standards for software development. This resulted in 1995 in the development of Unified Modelling Language (UML), a series of notation techniques for modelling software systems. In 2001 the Model Driven Architecture (MDA) framework was adopted. MDA is 'an approach to using models in software development' ([22] page 2-1).

In May 2009 the Ontology Definition Metamodel (ODM) specification 1.0 [23] was released. The ODM and its relation to MDA is described in paragraph 4.3.

## 4.1 Model Driven Architecture

The three primary goals of MDA are: portability, interoperability and reusability. In other words: to design a software system without worrying which environment it has to run in. To achieve this goal, MDA provides a modelling approach and enables software development tools [22].

---

<sup>4</sup> <http://www.omg.org/>

A *system* in MDA terminology may include anything between a software component and a whole enterprise environment. A *model* is a description or specification of the system and its environment. The *architecture* refers to the parts and connectors of the system and the way they interact.

The approach is identified as *model-driven* ' because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification.' ([22], page 2-2)

MDA begins with specifying the requirements of the system in a representation that is independent of any computing environment. This model is called the Computational Independent Model (CIM). In this process usually subject matter experts that will be using the software system are highly involved. Usually the CIM is a high level representation of business processes or procedures.

A software architect then creates a Platform Independent Model (PIM) based on the CIM. The PIM is a model that shows the functions and data structure of the future system but does not have any technical relation with the environment it has to run in. Next the software architect selects a system operation platform. This selection is usually based on both system requirements and the organisation's IT standards.

MDA provides tools, guidelines and specifications to translate the PIM into a Platform Specific Model (PSM). This is *model transformation* is a very important concept of MDA.

### **Model transformation**

A *mapping* determines how the PIM is transformed into the PSM. In complex systems, different parts of the PIM can be mapped to different PSMs that together represent the original PIM. To achieve this, particular parts of the PIM are *marked* for transformation. The results of transforming a PIM into a PSM are the PSM itself and a record of transformation. The record of transformation shows which element of the PIM is mapped to the corresponding elements of the PSM. Some tools like the Eclipse Modelling Framework can transform a PIM directly into deployable code, such as Java, C# or Data Definition Language (DDL). The latter is used to generate a database structure. We can therefore say that a model can be *initial* (typically a PIM) or *derived* (typically a PSM).

## **4.2 Unified Modelling Language**

MDA models can be created using UML. UML stands for Unified Modelling Language. It is a standard modelling language for visualising, specifying, and documenting software systems. Both PIM and PSM can be constructed using UML, since the language covers various levels of abstraction. UML 2.0 defines thirteen types of diagrams, divided into three categories: static application structure (six types), general types of behaviour (three types) and other aspects of interactions (four types). [3]

In the context of this thesis two diagrams are important: the structure diagram *class diagram* and the behaviour diagram *use case diagram*.

The class diagram represents the main concepts of a computer system with their relations, operations and attributes (see Figure 8).

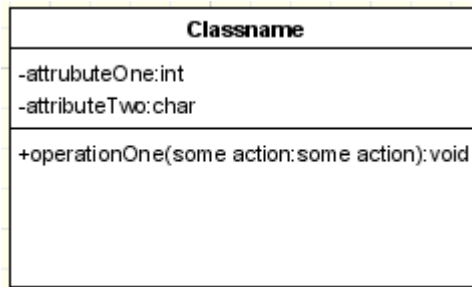


Figure 8: Example of a class diagram

Using the transformation model for DDL a class diagram can be translated into a database structure. The class name will thus be transformed into the table name and the class attributes into the column values with their data types.

From a database interface prompt the database structure can automatically be generated using the DDL code. Just like Java or C#, DDL can be generated by many MDA tools.

The use case diagram is used for capturing the user requirements. It is represented as a drawing of use cases (sequence of actions), actors (person, organization, or external system) and associations (the relation between actor and use case). An example of a use case diagram is given in Figure 9. The stick figures are the actors, the horizontal ellipse is the use case and the connecting lines are the associations [24].

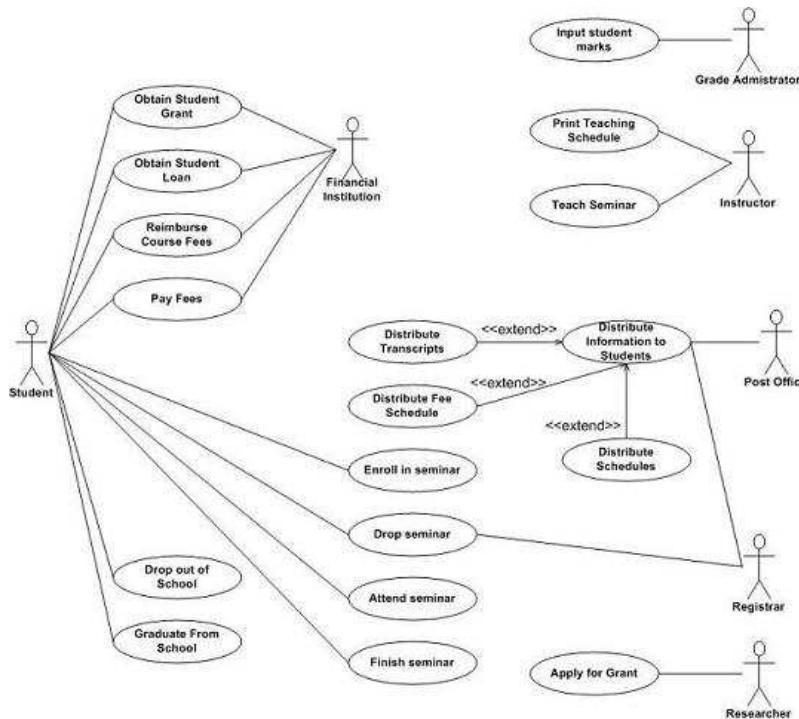


Figure 9: An example of a use case diagram, taken from [24]

## 4.3 Ontology Definition Metamodel

The Ontology Definition Metamodel (ODM) specification can best be introduced by citing a part of the scope of the ODM ([23] page 1): ' This specification represents the foundation for an extremely important set of enabling capabilities for Model Driven Architecture (MDA) based software engineering, namely the formal grounding for representation, management, interoperability, and application of business semantics.' This citation indicates that the OMG acknowledges the added value of AI techniques to the MDA. AI technologies introduce languages for formalising (grounding) concepts as well as methods for reasoning, validating and consistency checking.

The ODM is defined as a metamodel (model of models) and consists in itself of six metamodels: four that are normative, and two that are informative. The normative metamodels are:

- Formal logic languages (Common Logic (CL) and DL). CL is referred to as the first-order predicate language, (DL is non-normative)
- RDFS (RDF Schema),
- OWL and
- TM (Topic Maps)

RDFS, OWL and TM are commonly used in the semantic web community for describing vocabularies, ontologies and topics respectively.

The informative metamodels are:

- UML and
- ER (Entity Relationship, a representation of data used for database modelling)

These two are described as modelling languages particularly used for conceptual or logical modelling. These models are informative in the ODM because they are described in other OMG standards (UML2) or 'expected to be provided' (ER diagramming) ([23] page 31).

In addition there are mappings defined for transformation of UML and Topic Maps to and from OWL. This allows the integration of different levels of concept abstraction in the software development process.

## 4.4 UML versus OWL

Since the ODM adapts both UML and OWL the OMG clearly indicates that however they are both languages for modelling they must be considered complementary and intended for different purposes. This is also the conclusion of the authors of [25] who compared OWL and UML for the modelling of disaster management processes. The

authors here indicate that however the two use the same elements (classes, attributes and relations) the implementation of these elements in the language is very different. To name a few:

- OWL is property centric and UML is class centric. This has various consequences such as: in OWL a property can exist without a class, this is not possible in UML. In OWL a sub-property is a valid concept, it is not in UML
- individuals are a necessary part of an OWL ontology, they are not part of a UML class diagram (the term instance is used in UML). In an UML context, instances exist in runtime and typically change value while the program is running. The UML Object diagram does work with instances however, but this diagram is intended mainly to visualise how the classes would interact in runtime.
- in UML the class behaviour can be modelled by using operations. In OWL this must be done by specifying it as a property. There is no distinction in the OWL language however to indicate if a property refers to behaviour or some value.
- UML is based on the Closed World Assumption and OWL is based on the Open World Assumption

Both OWL and UML can be part of a software development process. This is outlined in the work of Kroha and Gayo [26]. These authors have investigated how ontologies can be used for the overall process of System Requirements Specification (SRS), of which modelling is a sub task. The authors advocate the use of ontologies in the SRS because ontologies store the explicit formal specifications of the concepts and relationships used in the application domain. The authors propose to use the application domain ontology as a basis for the communication in the SRS process. The authors also describe how a UML model that was created during the SRS process can be converted to OWL in order to be compared with the domain ontology. As such a 'reality check' can be executed. They propose to use semantic tools like reasoners and rule engines to check the completeness, correctness and consistency of the two ontologies. In particular they advise to use Protégé in combination with the RACER reasoner and Jess for a rule engine. More specific, JessTab is mentioned because it integrates the ontology environment of Protégé with the rule engine Jess.

Because the instrument that was to be designed in the thesis research can be considered some kind of a system requirement definition tool the selection of the tools used in the research work was based on [26].

This chapter has outlined which methods and technologies can be used for the creation of the research product. It has shown that the sciences of AI and Computer Science have integrated to a certain extent. This will be used for the creation of the research product. We will use these methods together with concepts of Geo Information science as described in chapter 2.

## 5 Building the Prototype

This chapter describes which tools were used for the production of the research end result and why they were chosen.

Regarding the objective of the research the literature study has learned that there are many possibilities available for creating the advisory instrument. The requirements that were previously listed in paragraph 1.2.3. can now be rephrased using the insight achieved from the literature research:

1. the instrument must be capable of *storing and retrieving existing knowledge* of
  - spatio-temporal solutions (such as DBMS, data type, access method, data model) and
  - characteristics of spatio-temporal (sensor) data
2. the system must be able to *acquire knowledge* by having the user of the system entering information about the user requirements (functional and non-functional) and about the data set he wants to work with
3. the instrument must be *flexible enough* to adapt new knowledge
4. the instrument must be able to automatically guide the user to the best solution to his or her needs by *reasoning* on the knowledge and user entered information and *inferring* conclusions
5. the instrument must provide a *ranking order* for all the presented solutions
6. where possible the instrument must *transform* the model by *generating code* to automatically create the database design

From what was learned in the literature study (documented in chapter 3 and 4) we can now motivate the selection of the main modelling language, UML or OWL.

Ad 1)

To store *existing knowledge* requires the use of instances in UML or individuals in OWL. These two terms have the same meaning, namely the implementation of a class in an *existing* item. As was noted in paragraph 4.4 individuals are present in an OWL ontology and they are not present in an UML class diagram.

Ad 2)

The system must provide a means to the user of the sensor data to enter requirements. These requirements are necessary to determine the solution, so they are essential to the system. The user requirements are existing features which makes the Ad 1) also valid here.

What about the UML *use case diagram* ? This is a requirement specification instrument that operates on the abstraction level of the user. This is true, but it is not the *user itself* who enters information, it is the system architect that creates the use case diagram based on information provided by the user. As a contrast using an OWL ontology editor, we can create a form where the user itself can enter information to the system without the presence of the system architect.

Ad 3)

'The instrument must be *flexible enough* to adapt new knowledge'. Ad 1) already indicated that knowledge as a real existing phenomenon can best be modelled with OWL. Since ontologies are written in OWL, an open XML-like syntax, they are *designed* to be connected to other ontologies to interoperate. So by connecting ontologies via web based interfacing we can guarantee the flexibility to adapt new knowledge.

Ad 4)

'The instrument must be able to automatically guide the user to the best solution'. This can be done by programming code that runs on instantiations of UML classes. This then would happen in runtime.

Using OWL we already have individuals (they do not have to be instantiated in runtime such as with UML) on which we can run a reasoner. When we use a rule-engine we need to write program code, as with UML. The individuals are accessible to the rules as soon as they become facts. This individual-fact conversion could in a sense be compared with the instantiation of a class in UML.

Ad 5)

At this point in the research it is not clear which model language provides the most appropriate ranking method.

Ad 6)

UML tools can generate code based on the created *class diagrams*. This is based on the idea of model transformation and is an essential characteristic of UML. Regarding OWL we have learned that a rule engine is a programming language that can use the individuals in OWL and 'do things' with them. At this point in the research we do not know how this will work out with generating DDL code using the individuals.

It are mainly the capabilities to store existing knowledge and adapt new knowledge, to provide a real user interface and to work with (reason on) individuals that have led to the decision to build the prototype as an OWL ontology. The selection of the tools Protégé and JessTab has been based on the work of [26], as described in paragraph 4.3. To discover how and whether the requirements listed above can be implemented in the prototype is an important subject of the research. The results will be summarised and discussed in the conclusions and recommendations in chapter 9.

The next paragraphs describe how OWL was used to build the prototype and how the tools helped to accomplish this.

The prototype was built using the ontology editor Protégé and the rule engine Jess. The reasoners Pellet and RACER were evaluated.

## 5.1 Protégé

The wiki website of Protégé <sup>5</sup> describes the product as follows: “Protégé is a free, open-source platform that provides a growing user community with a suite of tools to construct domain models and knowledge-based applications with ontologies.” Of the available ontology development environments this particular ontology editor has been selected because it is well documented, free of charge and it supports many plug-in environments.

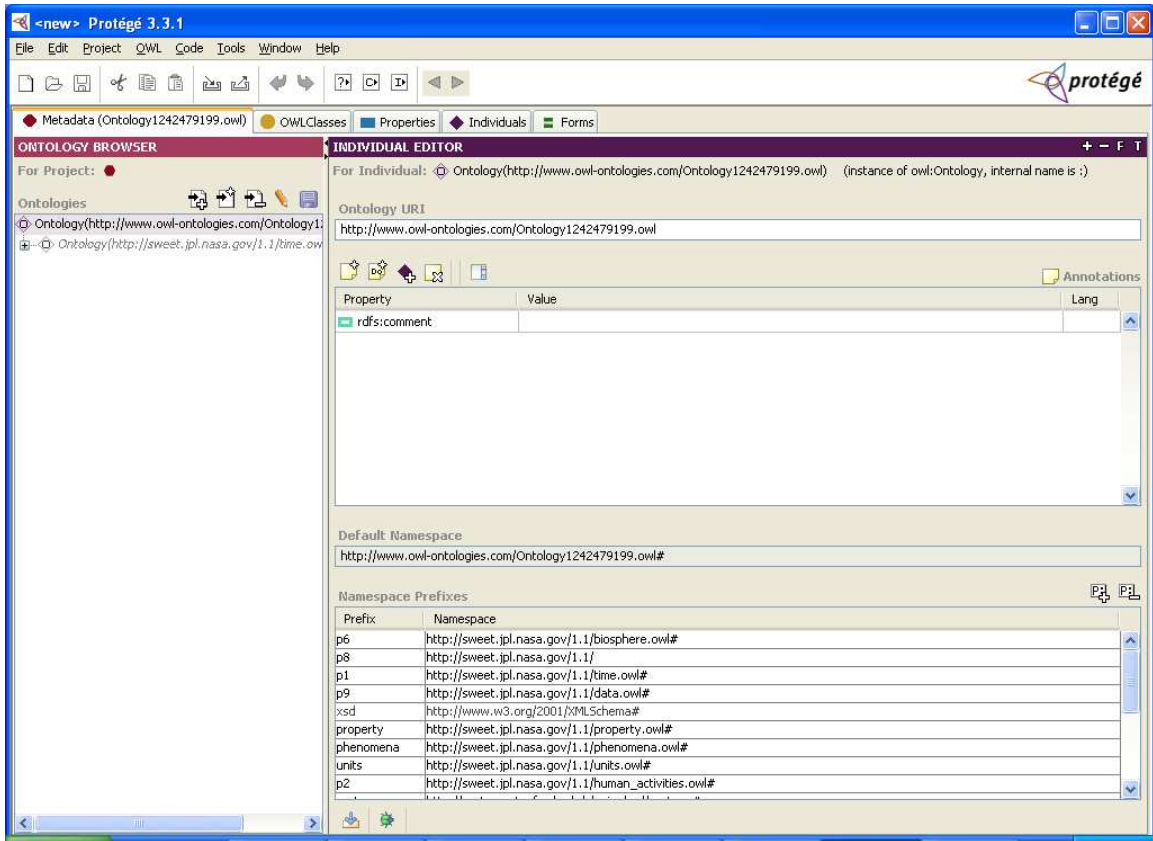
In this research version 3.3.1 is used even though it is slightly outdated. This was necessary because JessTab is not compatible with in higher versions of Protégé. This did not affect the building of the ontology in any other way.

The next five pages show screenshots of Protégé and its functions.

Protégé supports different semantic languages such as RDF Schema and various OWL versions. In this research the OWL-DL environment was used because of the requirement to support reasoning.

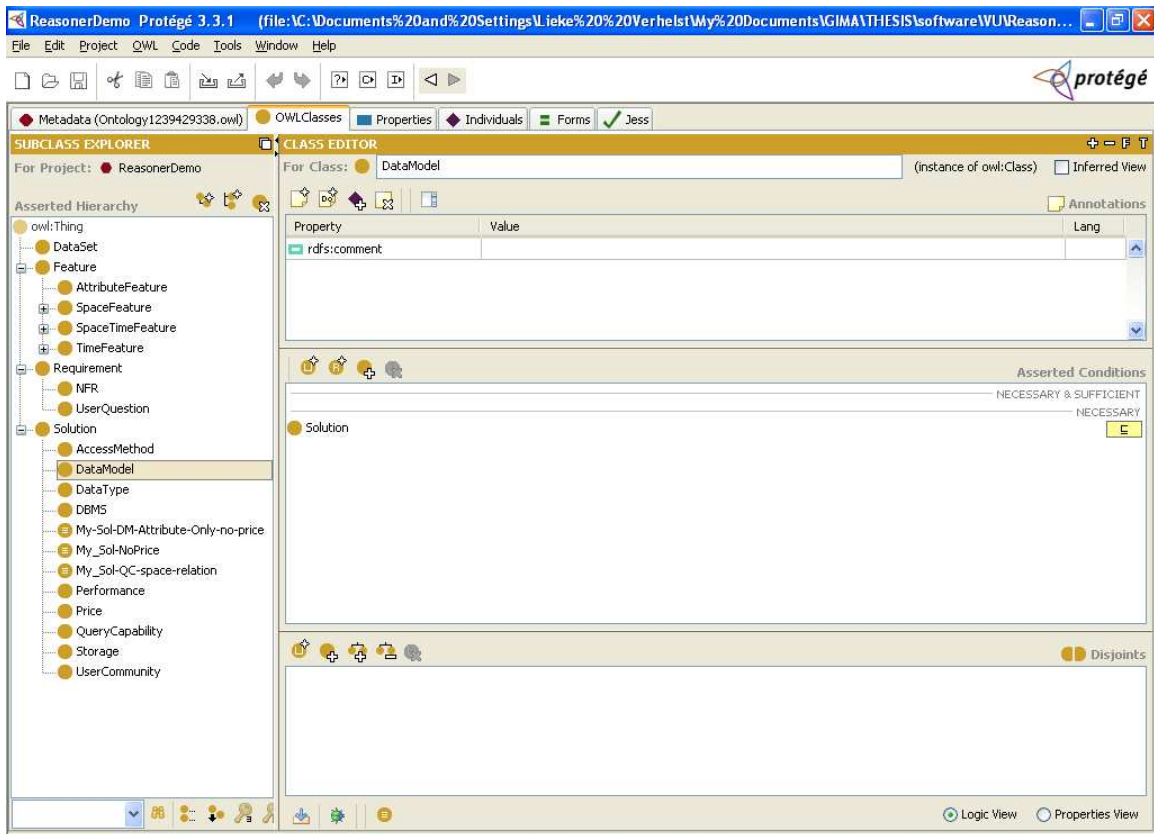
---

<sup>5</sup> [http://protegewiki.stanford.edu/index.php/Main\\_Page](http://protegewiki.stanford.edu/index.php/Main_Page)



**Figure 10: Protégé ontology browser**

This opening screen of Protégé allows the user to connect to existing ontologies, either on the Internet or to a file stored on a disk.



**Figure 11: Protégé Class editor**

This is the Protégé interface where the classes are defined. Classes are typed in manually and the class names can be changed any time. Subclasses are created as a child of the super class. The overall system parent class is the class Thing.

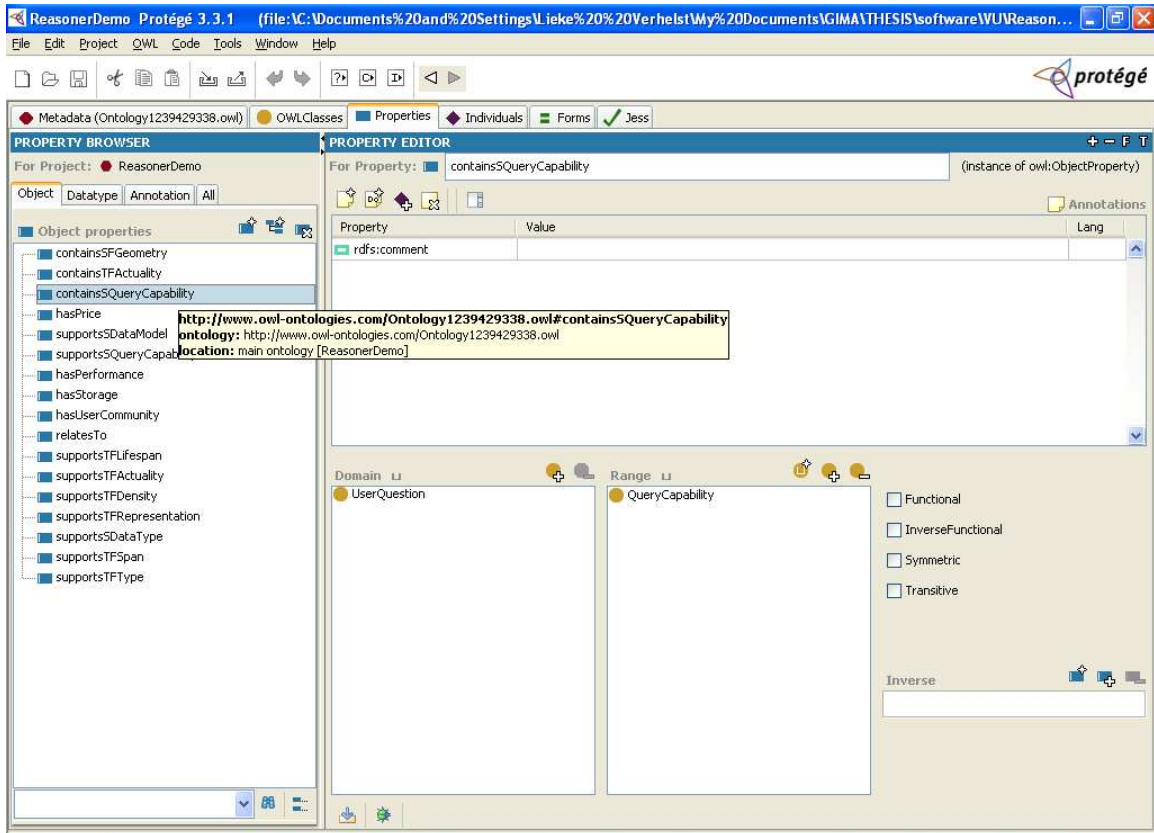


Figure 12: The Protégé Property Editor

In this editor the Protégé properties (object and data type) are entered. A property can have a domain and a range, these define the scope of the property. In the case of the example of Figure 12 one should read (like a O-A-V triple):  
 UserQuestion *containsSQueryCapability* QueryCapability

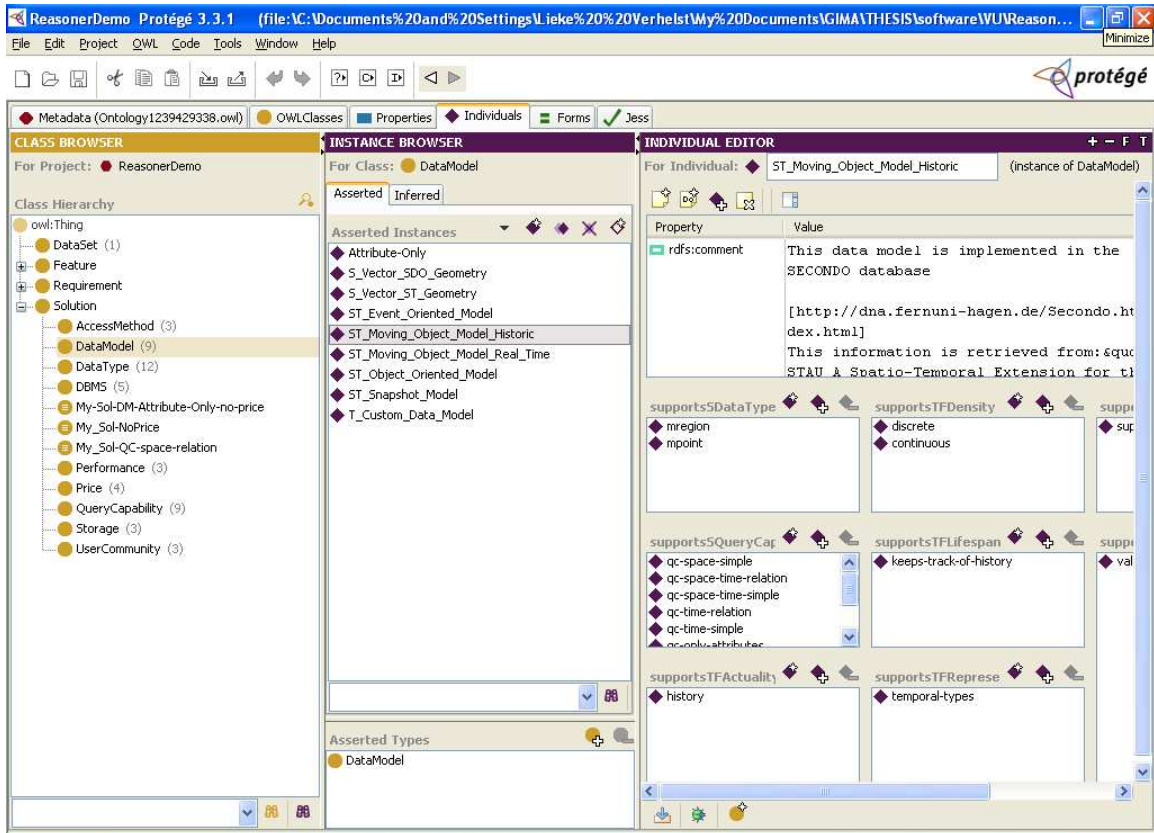
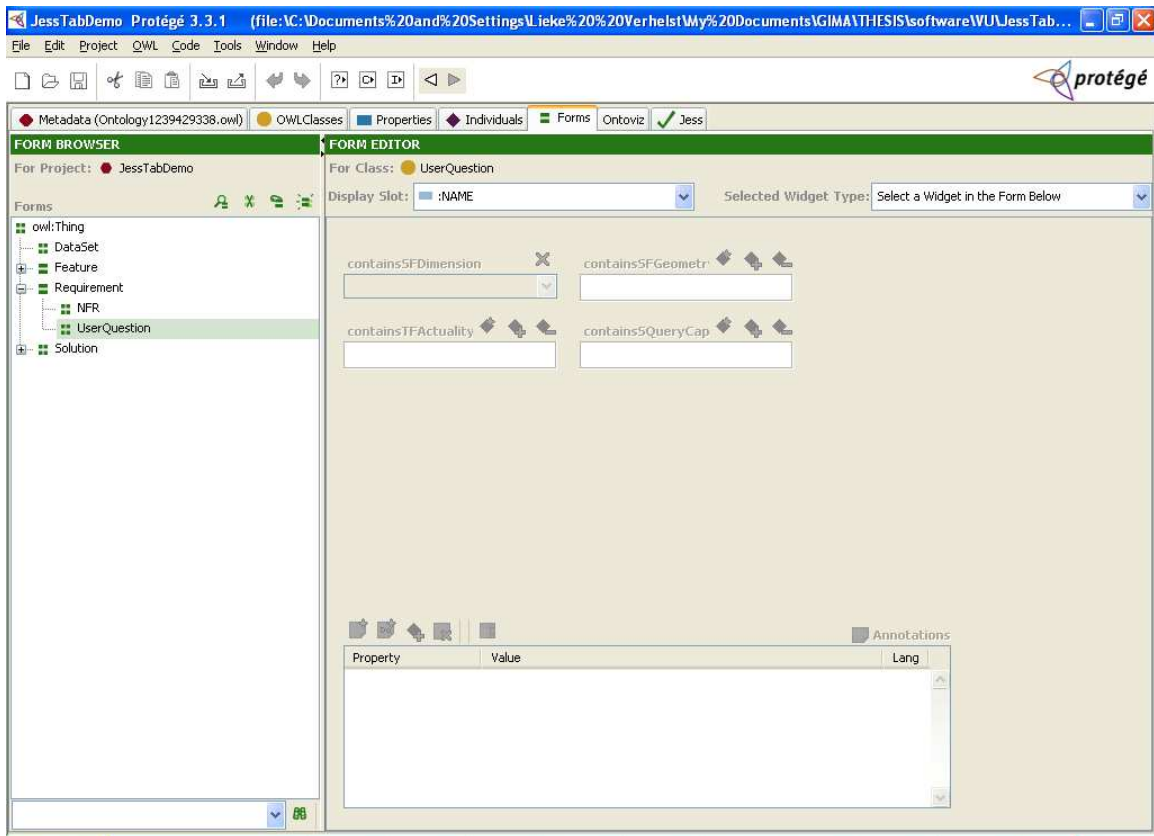


Figure 13: The Protégé Individual editor

In the Protégé individual editor individuals can be created for classes. Created individuals are called asserted individuals in contrast to inferred individuals who are identified by the reasoner. How many individual a class has is listed with a number next to the class name. When inferred individuals exist two numbers are shown next to the class name, one for the asserted and one for the inferred individual. This is not visible in Figure 13, but will be shown later on in Figure 34).

An individual has properties, these can be filled in as fields in a form.

Here is where the existing knowledge of a) existing spatio-temporal solutions and b) spatio-temporal features is entered.



**Figure 14: The Protégé Form editor**

The Protégé Form editor determines the layout of the individual form. It determines where and how the fields are located on the form.

This outlines the user interface that can be used by the user of the sensor data. As can be seen in the screenshot, the user can enter the following information regarding the questions he/she would pose in the data here:

- the data dimension
- which geometry is queried (point, line, polygon)
- whether the data is streaming (real time) or a static file (historical)
- which type of query (the query capability as described in paragraph 2.4) the user wants to pose on the dataset

There is also a user interface to enter information about the data set:

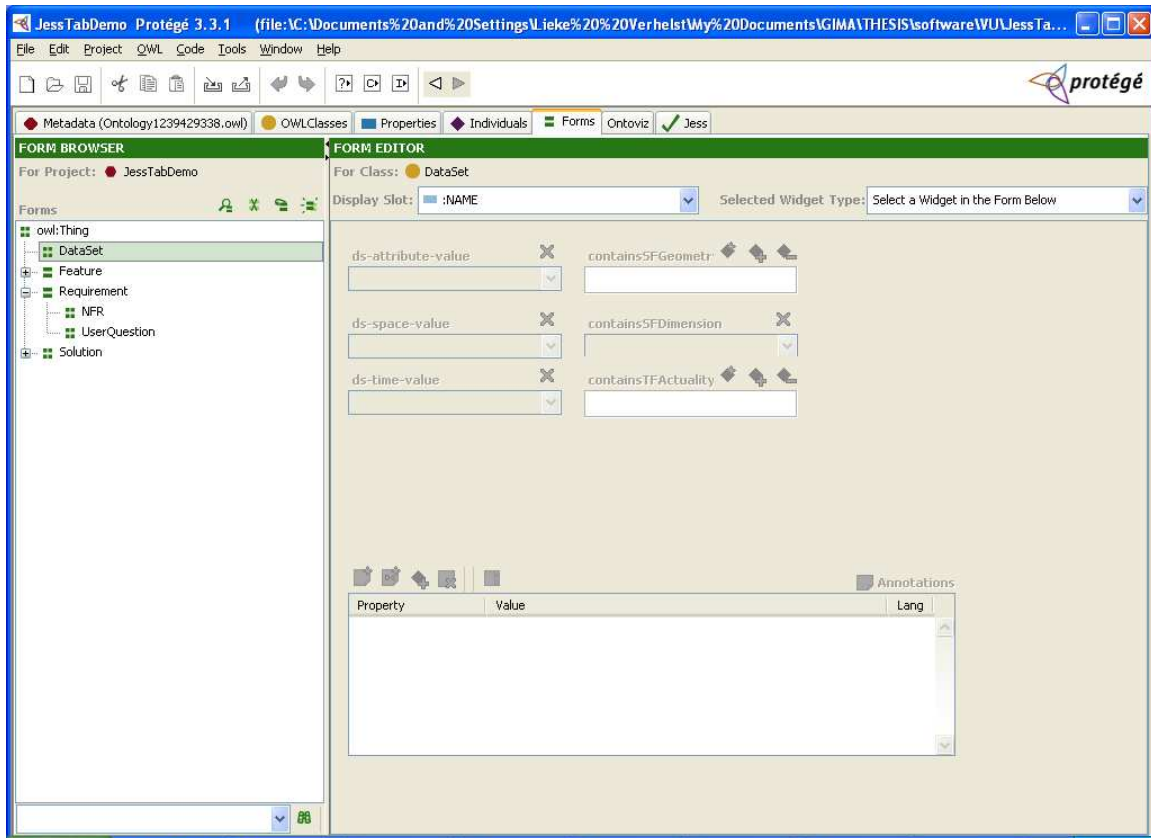


Figure 15: The user interface for entering information about the data set

As can be seen in Figure 15 the user can enter the following information:

- how/if the attribute value is stored in the dataset (values: 'attribute as value in record' and 'no value in record')
- how/if the space value is stored in the dataset (values: 'space by coordinates in record', 'space by identifier in record', 'space by identifier in data file name' and 'no value in record')
- how/if the time value is stored in the dataset (values: 'time by timestamp in record', 'time by identifier in record', 'time by identifier in data file name' and 'no value in record')

These three above are important for determining whether the structure of the data file is sufficient for obtaining the required solution after storage in the database. For example, when the user wants to execute a spatial query and the space value is not stored as coordinates in the file, but with an identifier like 'sensor station North' this information alone is not sufficient to exercise a spatial query. The spatial coordinates must be available for the query as a spatial object created from the spatial coordinates. In this

example this can be achieved by pre-processing the data by executing a lookup for the coordinates from a table that has the coordinates stored with the identifier ('sensor station North'; 59,54 N : 10,44 E ).

- which geometry is stored (point, line, polygon)
- of which dimension
- whether this is a streaming file (real time) or a static file (historical)

## 5.2 Jess

Jess<sup>6</sup> is a rule engine based on the syntax of C Language Integrated Production System (CLIPS). CLIPS was originally created to support expert systems. It was developed by the Artificial Intelligence Section of NASA in 1984 [27].

Jess is developed at Sandia National Laboratories<sup>7</sup> in the late 1990s. It is entirely written in Java, so it can be used in any Java supported environment. Jess can be licensed commercially, under an academic license or used with an evaluation license.

A Jess rule consists of a left-hand side (LHS) containing the conditions to be evaluated, and a right-hand side (RHS) separated by a '='>' sign. The RHS stores the procedures that are to be executed whenever the LHS evaluates to TRUE.

Every command is encapsulated between brackets '()'. Statements are executed with the operator listed first. '(+ 2 3)' means 'add two and three and display the result.'

Rules are declared using the (*defrule...*) statement.

When a rule is declared it is not immediately fired. It is just stored in memory. Rules are evaluated whenever the (*run*) command is given.

Rules run on information that is stored as **facts**. A fact is stored with a specific data format using the (*deftemplate*) statement – to define the data structure- followed by the (*assert..*) statement. An example for storing the fact that John is a person of age 23 is [28]:

```
(deftemplate Person "this is a template of a person"  
  (slot name)  
  (slot age))
```

---

<sup>6</sup> The name Jess is not explained on the Jess homepage [www.jessrules.com](http://www.jessrules.com). It is assumed that the 'J' is for Java and the 'ess' is for expert system.

<sup>7</sup> <http://www.sandia.gov/about/index.html>

```
(assert (Person (name "John Doe")
                (age 23)))
```

The term 'slot' is comes from RDF terminology. It is a synonym for a database column or to a 'property' in OWL.

An example of a rule that can run on this fact is:

```
(defrule twenty_one
  (object (is-a Person)
   (name ?n) (age ?a&:(>= ?a 21)))
=>
(printout t "The person " ?n
          " is 21 or older" crlf))
```

The rule determines if there is a fact of class Person that has a value in property 'age' that is larger than 21. It then prints the value of property 'name' in a sentence.

When a rule is trying to act on something that is not stored as a fact, the rule does not fire. This is the implementation of Jess working according to the closed-world assumption<sup>8</sup> (CWA).

## 5.3 JessTab

JessTab was written by Henrik Eriksson of the Linköping University in Sweden<sup>9</sup>. It is intended to create a bridge between Protégé and Jess. It allows mapping of Protégé knowledge as facts into Jess. With this combination a rule based expert system can be developed.

JessTab is a plug-in for Protégé, it has to be enabled and installed. (Details are listed in appendix 2). Once installed, several JessTab tabs can be seen. One is for the JessTab console to enter the Jess code and others are for storing and visualising the facts and rules.

---

<sup>8</sup> <http://www.nabble.com/JESS%3A-equivalent-rule-of-a-prolog-rule--td9192864.html#a9195102>

<sup>9</sup> <http://www.ida.liu.se/~her/JessTab/>

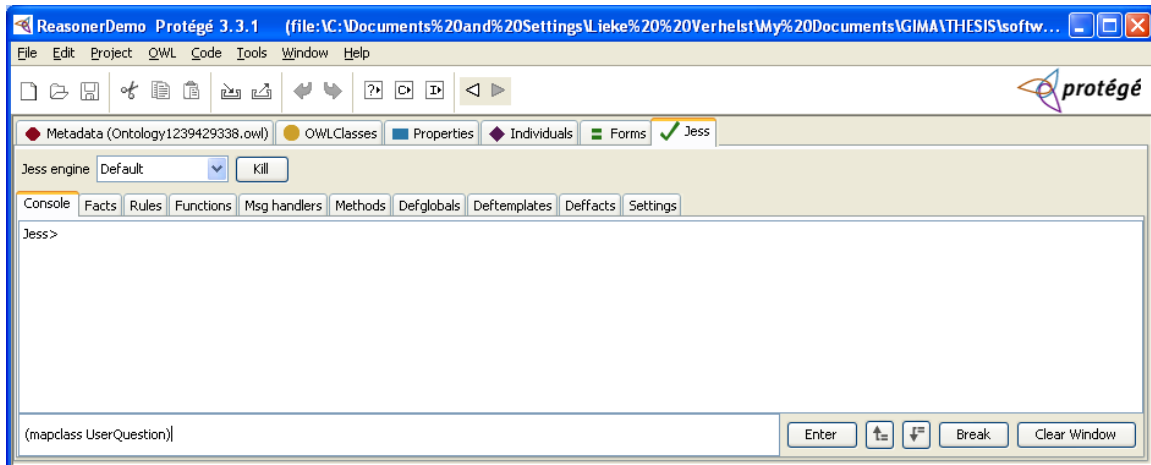


Figure 16: The JessTab interface in Protégé

With the command (*mapclass..*) individuals in Protégé are converted to Jess(Tab) facts using a Jess template called 'object'. The object template is the template created by JessTab to specifically map Protégé instances to Jess facts. This template definition (in JessTab called *deftemplate*) consists of the following default slots (properties):

```
(deftemplate MAIN::object
  "$PROTEGE-OBJECTS$"
  (slot is-a (type SYMBOL))
  (slot is-a-name (type STRING))
  (slot OBJECT (type OBJECT))
  (multislot rdfs:label)
  (multislot owl:versionInfo)
  (multislot rdfs:comment)
  (multislot rdfs:member)
  (multislot :NAME)
  (multislot rdfs:isDefinedBy)
  (multislot rdfs:seeAlso)
  (multislot owl:differentFrom)
  (multislot owl:sameAs)
  (multislot rdf:value)
  (multislot protege:inferredType)
  (multislot rdf:type))
```

These slots are complemented with the property values of the class in the ontology that were specified by the creator of the ontology as can be seen in the following example. In this example a very simple ontology was built from the O-A-V triple of Figure 1 (Ball-colour-yellow). The individual MyBall (class Ball, attribute colour, value yellow) is mapped to a fact. The result below shows the default template slots and the slot 'colour' that was created by the creator of the ontology:

```
(MAIN::object (is-a Ball)
  (is-a-name "Ball")
```

```

(OBJECT <Java-
Object:edu.stanford.smi.protege.owl.model.impl.DefaultOWLIndividual>
(rdfs:label )
(owl:versionInfo )
(rdfs:comment )
(rdfs:member )
(:NAME "MyBall" )
(rdfs:isDefinedBy )
(rdfs:seeAlso )
(owl:differentFrom )
(owl:sameAs )
(rdf:value )
(protege:inferredType )
(rdf:type <Java-
Object:edu.stanford.smi.protege.owl.model.impl.DefaultOWLNamedClasses>
(colour "yellow"))

```

It also shows that the slot value ‘:NAME’ has received the instance name (‘MyBall’) and how the ‘OBJECT’ slot has received a reference, not a value. This information is used frequently in the Jess code written for the thesis research.

## 5.4 The reasoners: Pellet and RACER

On the website of Protégé two reasoners are recommended: Pellet<sup>10</sup> and Renamed Abox and Concept Expression Reasoner (RACER)<sup>11</sup>. Pellet is mentioned for use in a Protégé course<sup>12</sup>. RACER is recommended for use in [20]. While Pellet is a simple open source reasoner, RACER is a commercial product. Both were used and compared here. They generated the same results for the simple tasks requested.

The reasoner must be started and the port it communicates on must be entered in Protégé to enable interfacing.

The reasoner is used here for two reasons:

- 1) check the consistency of the ontology
- 2) compute inferred types

---

<sup>10</sup> <http://clarkparsia.com/pellet>

<sup>11</sup> <http://www.sts.tu-harburg.de/~r.f.moeller/racer/>

<sup>12</sup> <http://protege.stanford.edu/shortcourse/protege/200703/prepare.html>

ad 1) This command was invoked via the Protégé OWL menu 'Check consistency'.

ad 2) this was invoked using the Protégé OWL menu 'Compute inferred types'. More details of how the reasoner is used with the prototype is described in chapter 8.

This chapter has described on which ground the selection of the tools was done. Also the basic working principles of the tools was outlined.



## 6 Building the ontology

In this chapter it is outlined from which information and how the ontology was built. It describes the issues that were addressed during this process.

The knowledge of spatio-temporal modelling that is stored in the ontology is primarily based on [8], chapter 2. This work contains a literature overview of spatio temporal data models. It lists important semantic descriptions of features of time and space and assigns these features to spatio-temporal models. This information is displayed in tables. The ontology that is created in this research is largely based on these tables.

Since the tables in [8] only contain information of spatio-temporal data models, it has been enriched in this research with other information taken from literature about relational [29], spatial [30] [31] and temporal databases [4]. In this way the prototype will also support users who have spatio-temporal data, but for some reason do not want to use the temporal or spatial dimension in their application. The thus enriched information is displayed in Table 1. The values in Table 1 that are not taken from [8] are the data models Standard relational model, SDO Geometry model, ST Geometry model and Temporal model and the column value Actuality. Sources of this information have been listed under the table. The Actuality column has been added to indicate that some models support also real time and future movement of objects (the real time Moving Object). Table 1 is referenced further in this document as ‘the spatio-temporal data model knowledge table’, or briefly, knowledge table.

Unfortunately we cannot determine if the information taken from the other sources can be interpreted in the *same way* as the information from [8], since this source does not indicate exactly how the classifications in the table have been made. Therefore the information in this table must be interpreted as an assumption.

Not everything of the table information of [8] was used for reasons of time constraint and since this would not add to the overall objective of building a working prototype. Only the columns that were considered as most likely to be used in a real life situation were used. In addition, it must be noted that no attempt has been made to make the ontology totally complete since this is a proof of concept situation.

**Table 1: The knowledge of the data model capabilities stored in the ontology**

Model name	Features of Space					Features of Time				
	Measurement	Topology	Dimension	Density	Lifespan	Span	Type	Actuality 6)	Representation	Query capability
Standard relational model 1)	No	No	No	n/a	n/a	n/a	n/a	History	Attribute of instance	1
SDO Geometry model 2)	Yes	Yes	2D, 3D	n/a	n/a	n/a	n/a	History	Attribute of instance	1,2,3
ST Geometry model 3)	Yes	Yes	2D, 3D	n/a	n/a	n/a	n/a	History	Attribute of instance	1,2,3
Temporal model 4)	No	No	No	Discrete, continuous	Yes	Yes	n/a	History	Temporal types	1,4,5
Snapshot model	No	No	All	Discrete	No	No	Valid time	History	Attribute of location	1,2,4,7
Event Oriented model	No	Yes	All	Discrete	Yes	Yes	Valid time	History	Attribute of an event	1,2,3,4,5,7,8
Object Oriented	Yes	Yes	All	Discrete, continuous	Yes	Yes	Valid time, transactional time	History	Attribute of object	1,2,3,4,5,6,7,8
Moving Object historic	Yes	yes	2D	Discrete, continuous	Yes	Yes	Valid time	History	Temporal types	1,2,3,4,5,6,7,8,9
Moving Object real time 5)	Yes	Yes	2D	Discrete	n/a	n/a	Valid time = transactional time	Real time, History, Future	Temporal types	1,2,3,4,5,6,7,8,9

1) Information taken from [29]

2) Information taken from [30]

3) Information taken from [31]

4) Information taken from [2] and [4].

5) Information taken from [7] and [8]

6) Information derived from [4], [31], [29], [30], [7], [2] and [8]

The n/a values could not be derived from available literature. In the time frame of this thesis it was not possible to conduct tests.

Numbered values for the query capability correspond to the following values:

1= attribute queries

2= simple spatial queries

3= relationship spatial queries

4= simple temporal queries

5= range temporal queries

6= relationship temporal queries

7= simple spatio-temporal queries

8= range spatio-temporal queries

9= behaviour spatio-temporal queries

Information included in the ontology about existing DBMS solutions that support either attribute, space, time or spatio-temporal features has been stored in Table 2.

**Table 2: Properties of DBMS included in the ontology**

DBMS name	Price	User community	Data Model
PostgreSQL	No price	Well supported	Relational
PostGIS	No price	Well supported	ST Geometry
Oracle Spatial	Large price	Professionally supported	SDO Geometry
Informix time series data blade	Large price	Professionally supported	Custom
Secondo	No price	Poorly supported	Moving object historic

The tables 1 and 2 are listed here explicitly because they are an important part of the ontology. The basic logic of the prototype is based on these tables. This logic is: the user requires a query capability, this query capability is supported by some data models and the data model is supported by some DBMS. This is how the prototype finds its DBMS solution.

The prototype contains much more knowledge and, because it is created as an ontology, it is designed for and intended to be extended by complementing knowledge. Currently the ontology contains knowledge (as ontology individuals) of:

- *solutions* (access method, data model (Table 1), data type, DBMS (Table 2), storage, performance, price, user community – the last four being non-functional requirements) and
- *features* of space, time and space time (Table 1)
- specifications of *data sets* (to be entered by the user)
- *requirements*, (to be entered by the user), functional (the user question) and non-functional

Figure 17 illustrates how the knowledge of data models, stored in OWL, is visualized through the Protégé ontology editor.

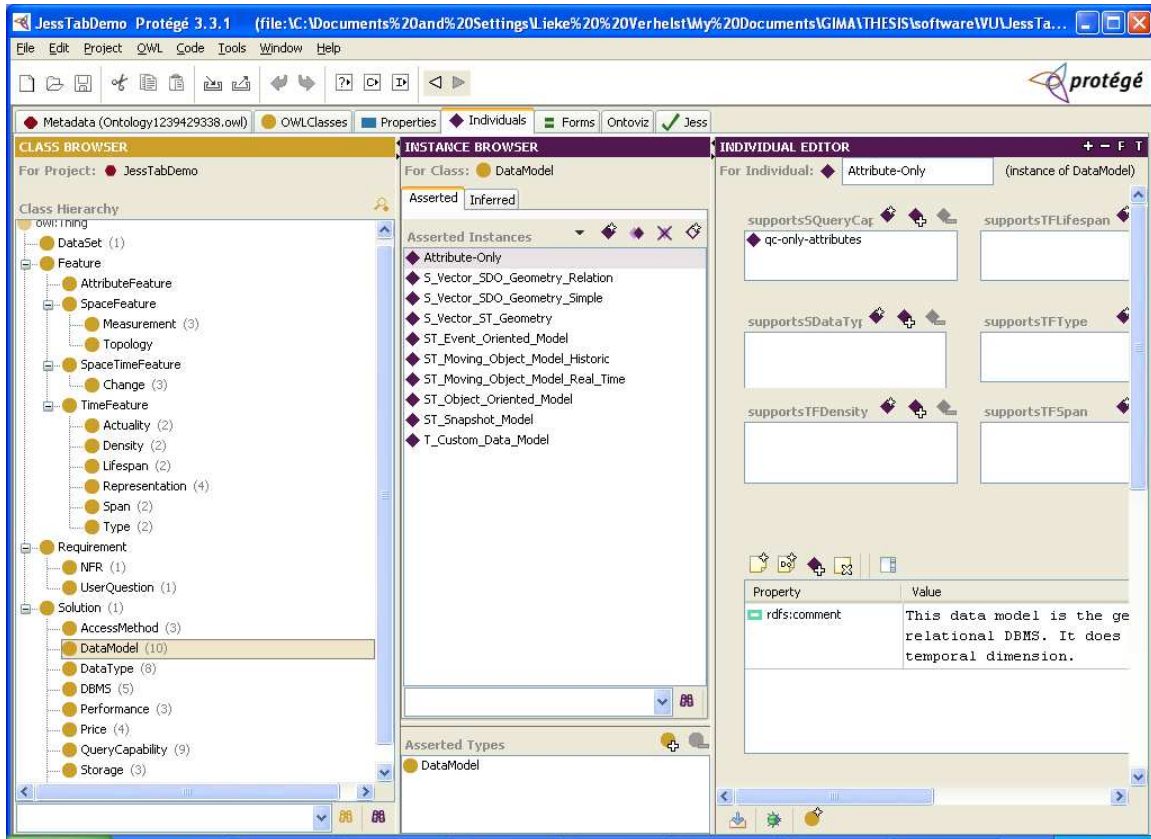


Figure 17: The knowledge of data models stored in the ontology

Not every piece of knowledge is used in the calculation of the solution for its intention is to be a prototype.

## 6.1 The base ontology

The creation of an ontology seems an easy job, however one is easily fooled by this. As indicated in [32] and [20] there is no one right way to develop ontologies. When relations between concepts can be described in many ways, consequently there are many possible ontologies for the concept. Fortunately there are some basic guidelines that can be used described in [32]. These have been followed in the process of the design of the prototype ontology (the design rules are printed bold below).

### **Consider the reuse of existing ontologies**

In an early stage of the research the Internet has been searched for ontologies for time and geography. There are some available, and it appeared that they are very different in terminology and structure. This is illustrated by the images of two ontologies from 'time' found on the internet (Figure 18 and Figure 19).

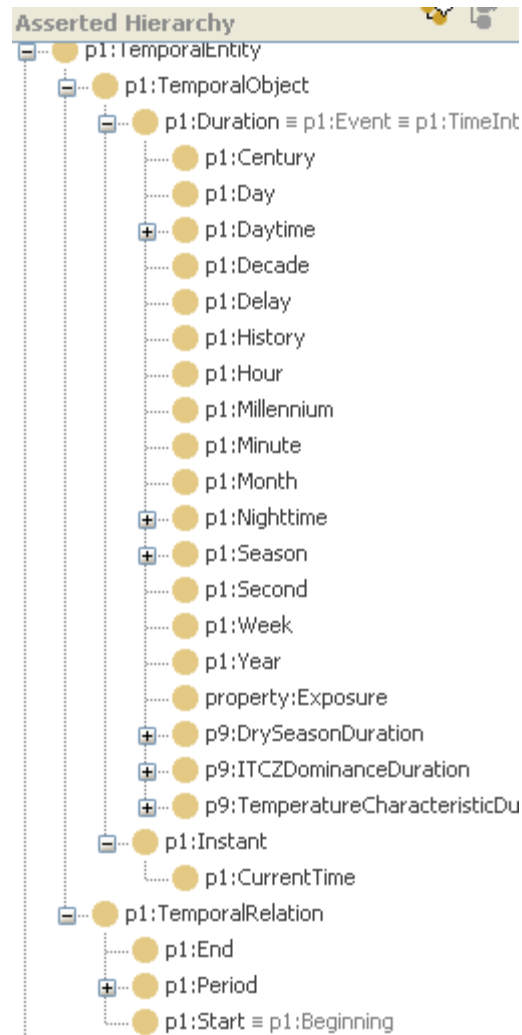


Figure 18: An ontology of time created by NASA, taken from [33]

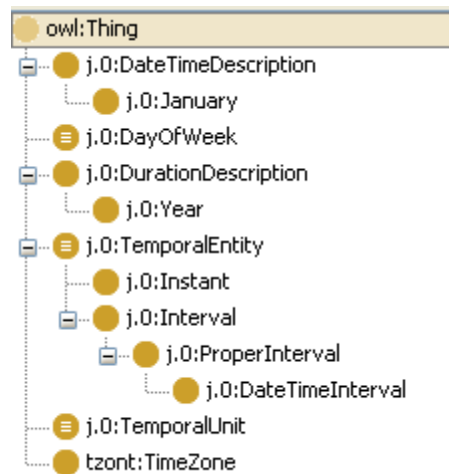


Figure 19: An ontology of time created by W3C, taken from [34]

This experience made clear that to create an ontology of not only time but also characteristics of space, time and space-time relations as well as their relation with existing solutions is a long lasting exercise and therefore not possible in the available timeframe. It was then decided to use the tables of [8] to create the ontology.

**Determine the domain and scope of the ontology, Enumerate important terms in the ontology**

The basic classes and properties were defined by determination of the essential classes and properties. To establish this, the general system requirement semantics were used: a Requirement determines a Solution. 'Requirement' and 'Solution' were defined as OWL classes, while 'determines' was defined as an OWL property. In the context of software engineering 'requirements' are often divided in 'Functional' and 'Non-Functional Requirements (NFR)'. This is adapted here by assigning the subclasses NFR and UserQuestion to the class Requirements, where UserQuestion relates to the functional requirements. The Solution class stores the available solutions such as DBMS, data model, data type.

In our case study two other important factors determine the eventual solution: the properties of the data set and the characteristics of space and time. For this reason the two classes DataSet and Feature were created. The relation between these classes and the Requirement and Solution class are depicted in Figure 20.

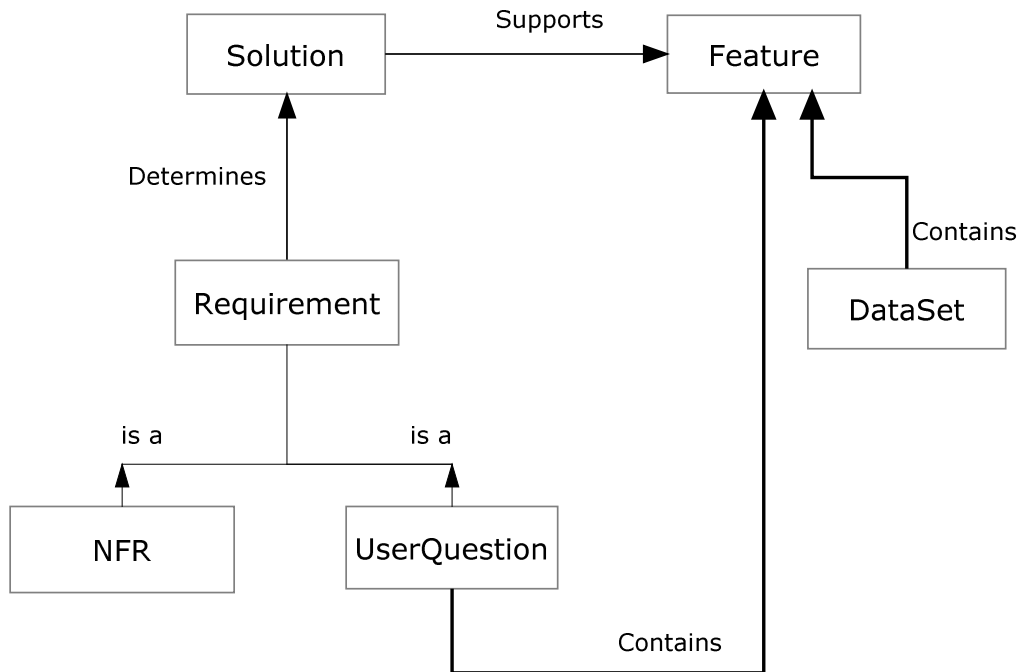


Figure 20: The basic classes and properties of the ontology

We can read this as:

'the user question contains features of space and time', 'the data set contains features of space and time', 'the solution supports features of space and time' and 'the requirements determine a solution'.

In other words: the ontology contains knowledge of **solutions** that *support features* of space and time in order to answer **user questions** about **data sets** *containing* space and time **feature** elements. The objective of the ontology system is to *determine* a **solution** that belongs to a set of **requirements** (classes are represented in bold, properties in italic).

Once entered in the Protégé ontology builder a graph can be generated of the classes and relations. This is essentially the same as Figure 20. It is repeated here in a different format to present how Protégé displays this information. Following graphs will be shown in Protégé format. The position of the classes and relations in the drawing are generated by the ontology drawing software<sup>13</sup> and do not have any hierarchical meaning.

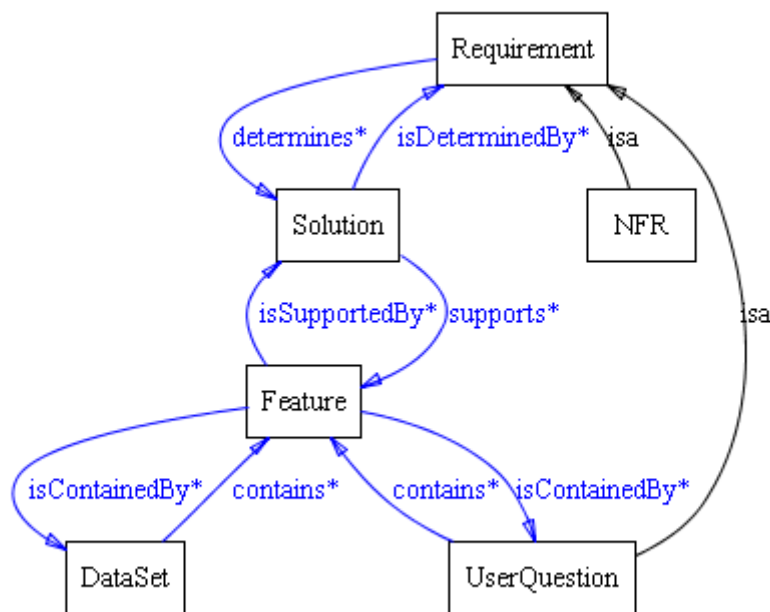


Figure 21: The basic relations between the model entities

<sup>13</sup> The drawings were generated by Ontoviz which is a Protégé plug-in. It needs Graphviz ([www.graphviz.org](http://www.graphviz.org)) to operate.

### Define the classes and the class hierarchy

The Feature class was subdivided in the features of space (class SpaceFeature), of time (class TimeFeature) and of space and time (class SpaceTimeFeature) to separate the semantics of these features. To include the feature 'value' (as in measurement value, for example 'temperature', 'humidity') the class AttributeFeature was added (see Figure 22). Note that the graphics in Protégé automatically show the subclass relation as 'isa' (is a).

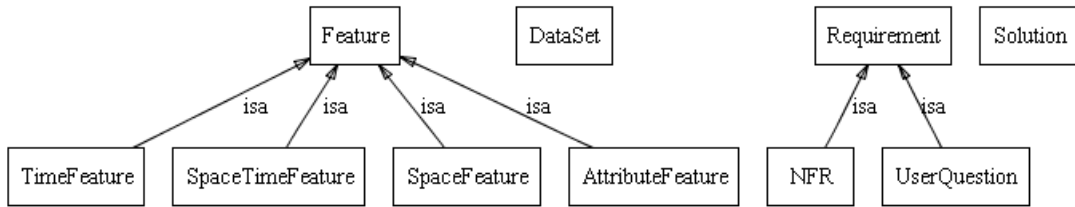


Figure 22: The Feature class with subclasses

After this new subclasses were added to describe (as sub-sub-classes) the semantics of attributes, space, time and space and time. This is shown in Figure 23.

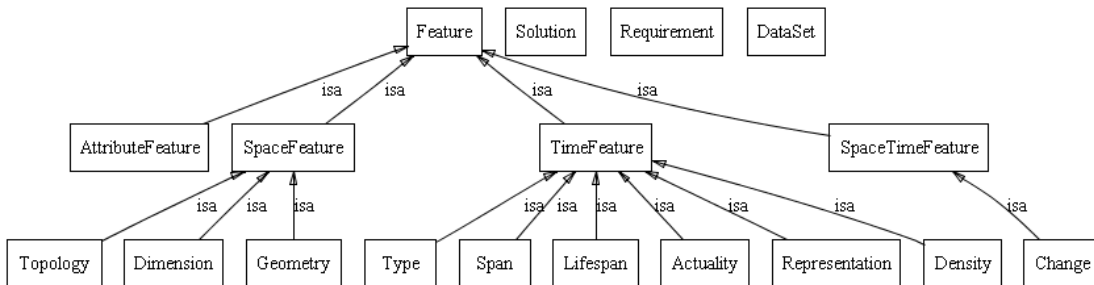


Figure 23: The Feature class with all the subclasses and sub-subclasses

The Solution class was extended by summing up database solutions for space and time storage as subclasses: (Figure 24).

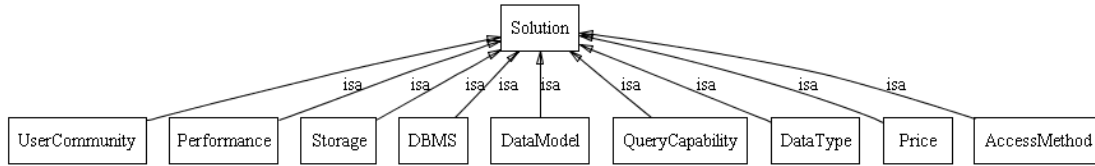


Figure 24: The Solution class with subclasses

When we depict the properties together with all the classes and sub-(sub-)classes we get something that is increasingly getting hard to read (Figure 25).

Note that we have removed the 'determines' property since this will eventually be replaced by a process (exercised by the reasoner or rule-engine) that takes care of this function automatically.

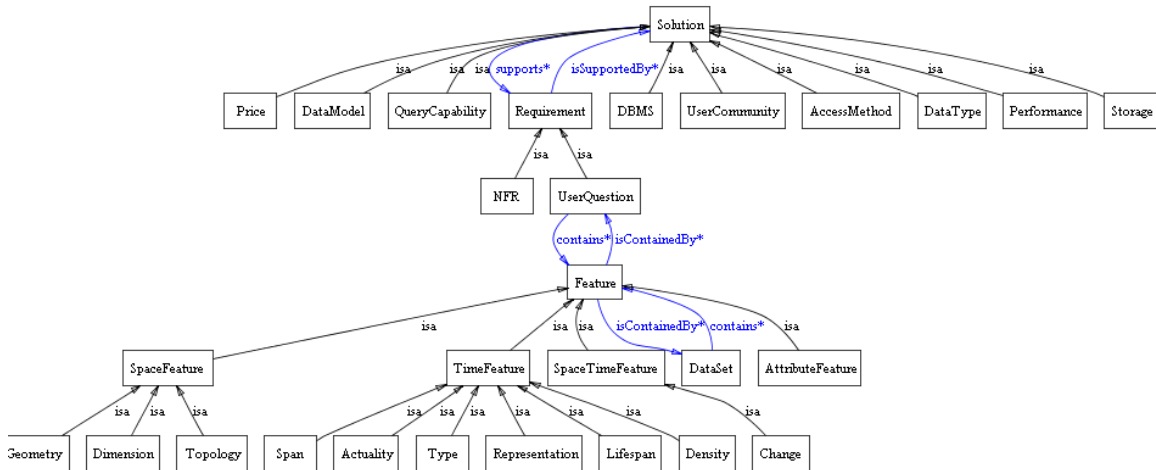


Figure 25: All classes and sub-sub-classes with properties

Figure 25 is not the end result of the ontology development process. How this is completed is described in paragraph 6.2.

## 6.2 Refining the ontology

After the basic design of the ontology was outlined it was adapted to the purpose of the case study. Some important requirements for the ontology were outlined already in paragraph 1.2.3 and chapter 5. They are actualised for the ontology development here:

- users must be able to enter information (typically user requirements and information of the data set) via a form
- since we have determined that a reasoner or rule-engine will take care of the *automatically guiding* of the user to the solution it must support the use of rules and facts
- its structure has to support programming the rule-engine
- its structure must represent a realistic semantic representation of the concept descriptions

In this process the contents of the knowledge table (Table 1) had to be translated into the ontology. It had to be determined which of the column and row information of the knowledge table is a class, subclass, individual or data property value. How this is done is important for the user interface. By using forms in Protégé it is possible to select values (individuals, data type properties) and connect them to a property in another class. This is the same principle as is used in a database application where in forms values can be selected from lookup tables. Several possibilities have been evaluated to determine which was the most appropriate. They are described below in an example to model the O-A-V relation between TimeFeature and Actuality and the two values for Actuality ('real time' or 'history'). The possibilities evaluated are listed in Table 3.

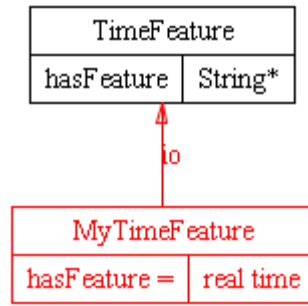
**Table 3: Mapping the contents of the knowledge table**

#	O	A	V1	V2
1	TimeFeature as class	hasFeature as data property	real time as data value	history as data value
2	TimeFeature as class	hasTimeFeatureActuality as data property	real time as data value	history as data value
3	TimeFeature as class	Actuality as subclass (with the is-a relation)	real time as instance	history as instance

The consequence of either choice has been outlined below.

### Example 1 and 2

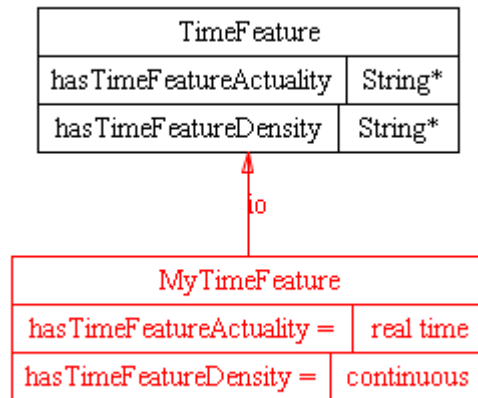
A data property 'hasFeature' of type string is created and connected to the TimeFeature class. This property is filled with the values 'history' and 'real time'. These values can be selected when the instance MyTimeFeature is created. This is illustrated in Figure 26 where the value 'real time' has been selected.



**Figure 26: A graphical representation of Example 1**

If we want to repeat this for all the features and feature values we cannot discriminate between the features and their values. The hasFeature property field would be filled with all the values for features of time, space and space-and-time without knowing to which feature (time, space, or spatio-temporal) the value belonged. As soon as the MyTimeFeature instance is created we are losing the information that the value 'real time' is in fact coming from the TimeFeature Actuality. This information is not stored in the individual, as is shown in Figure 26.

A solution for this is to create a property for each feature (hasTimeFeatureActuality etcetera) and fill this with single string values. This is the same as example 2 and shown as graph in Figure 27. This configuration would eventually result in a long list of properties, and results in a lot of redundant information (Figure 27).



**Figure 27: A graphical representation of Example 2**

It appeared that only to the individual level values can be selected in the Protégé form, not the levels below it. As a consequence the property values in the individual ('history' or 'real time') cannot be accessed via a form. In the context of the objective of the research this would mean that the user of the instrument can enter *some* information in

the system, however *not to the required extent*. Examples 1 and 2 were therefore disqualified.

### Example 3

To configure the ontology according to example 3 sub-sub-classes and individuals were created. The result is shown in Figure 28.

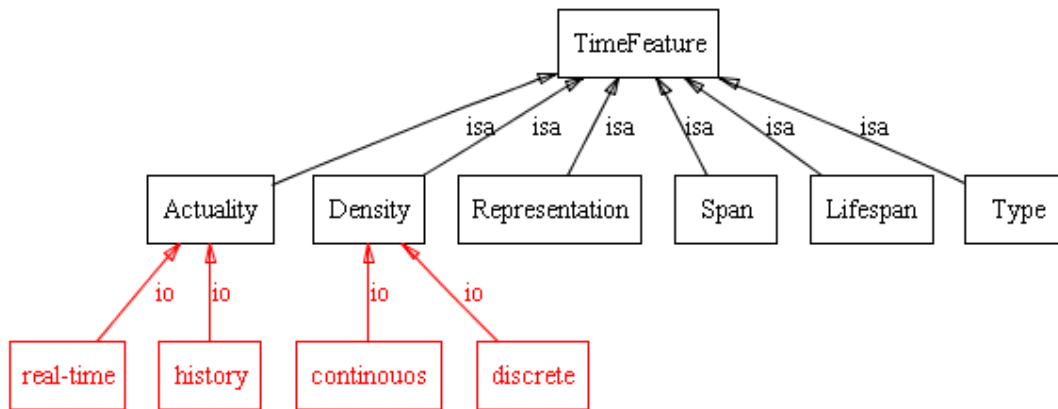


Figure 28: A graphical representation of Example 3

All the values stored in this ontology can be accessed via forms. This is why this ontology architecture was used and not the ones in example 1 and 2. Some other adjustments were made to the ontology to make the coding of rules easier. For example the property 'Dimension' was changed to data type instead of object type. Using a data type, logical comparisons could be coded like for example 'if dimension >2'.

The OWL code of the full ontology is given in appendix 3.

This chapter has indicated that building an ontology is a matter of carefully analyzing for which purpose it is intended. The next two chapters describe how one can work with the knowledge that is stored in an ontology. In chapter 7 the rule engine Jess is used for this, while in chapter 8 the results of querying an ontology with a reasoner is described.

## 7 Jess rules

In this chapter it is described how the knowledge that is stored in an ontology can be accessed and queried using a rule engine. Also the limitations of this method are addressed. The work in this chapter is complemented with the 'JessTabDemo' prototype and the Jess code software. The Jess code of is software is listed in appendix 2 and the ontology is listed in appendix 3.

As already outlined in paragraph 1.2.3 the objective is to automatically determine which solution fits best to the requirements that are entered by the user. The JessTab prototype supports the following solutions:

1. the creation of an individual YourSolution that contains a combination of property values of one or more individuals of the Solution class that is/are stored in the ontology. This result is displayed in Protégé.

and/or

2. a textual response in the JessTab console of
  - a. DDL code or
  - b. when the user enters information that is contradictory or technically impossible an advice how to proceed

An example of 1 is: the user specifies that he is going to query a time relation. The solution that the prototype will return is a set of property values displayed in Protégé (Figure 29). In this case: DBMS = Informix, data model = custom data model, cost= large price and user community= professionally supported)

An example of 2 is: the user specifies that he is going to query only attributes. The prototype will return a solution in Protégé comparable with the example above (with property values matching the user input, in this case DBMS=PostgreSQL) and DDL output in the syntax of the PostgreSQL DBMS solution (Figure 30)

Another example of 2 is when the user enters contradictory information that cannot lead to a solution, such as when the data file contains only two dimensions and the user asks for a three dimensional query. Or when the data file does not contain space coordinates and the user wants to exercise a spatial query. In these circumstances the user gets an advice from the prototype in text on the JessTab prompt.

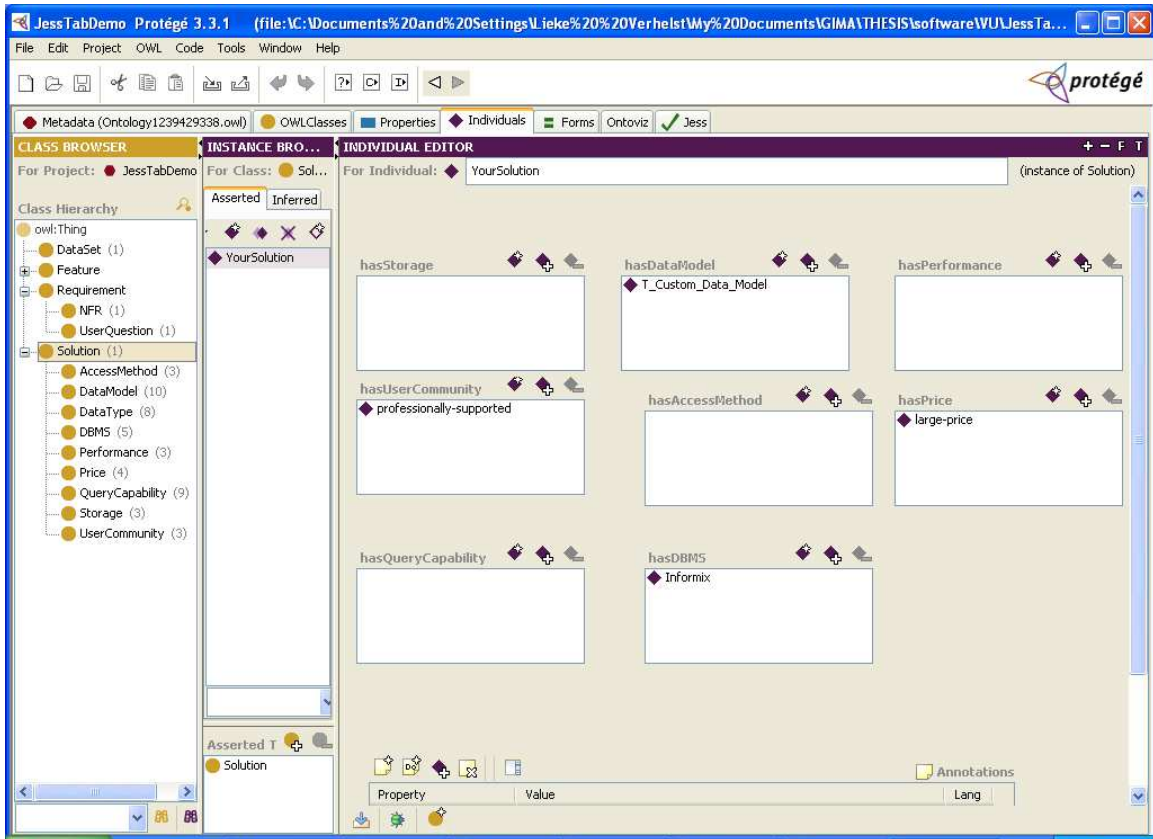


Figure 29: The result of the JessTabDemo as the YourSolution instance

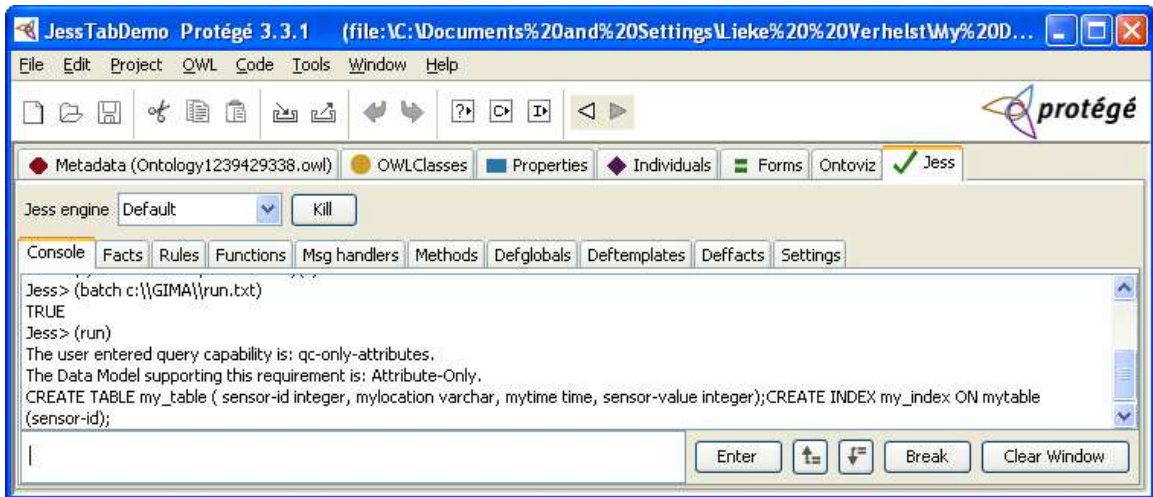


Figure 30: The result of the JessTab demo in the JessTab console

In paragraph 4.3 the rationale for using JessTab is explained by indicating that these tools were recommended in [26]. The current chapter describes which programmatic solutions were created for the objective. This was established using the following tutorials [21, 27, 28, 35, 36].

The base outline of the functions that needed to be implemented in the code were:

- a) define constraints that take care of correct combinations of features and solutions
- b) find solutions that belong to requirements
- c) provide a ranking mechanism to compare solutions
- d) guide the user to alternatives when a solution cannot be determined

ad a) This was enforced by the use of functions written in JessTab. One function was written as a principle example. This function prevents the user to find solutions for a space dimension higher than available in the dataset (if a data set contains information in 2D, one can never request a 3D solution). The name of this function is 'CheckIfDimensionCompatible' and it is listed in the Jess code in appendix 2.

ad b) It was assumed that for the user the query capability is the most important (and not for example performance or solution cost), therefore its value has been taken as the key to find the solution. The solution was determined following a sequence of rules:

**Clear all old rule and fact information.** This is done using the command `(clear)`

**Convert Protégé individuals to Jess Facts** by using the command

```
(mapclass <classname> )
```

The result of this command is the creation of a series of Jess Facts with variable values corresponding to property values as is outlined earlier in paragraph 5.3 :

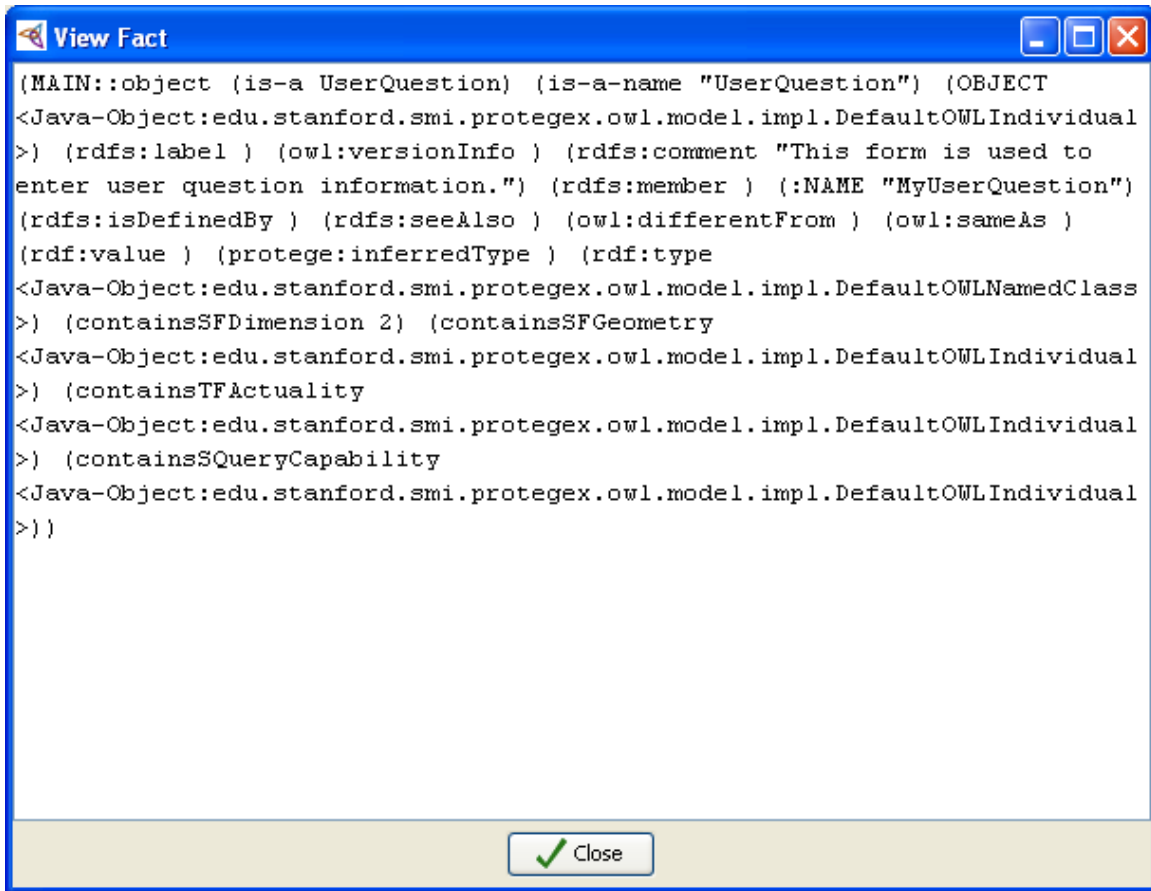


Figure 31: The UserQuestion individual converted to a Fact

Note that data value properties (such as containsSFDimension) contain values (in this case '2') and that individual properties (like containsSQueryCapability) store a pointer to the individual object

```

(<Java-
Object:edu.stanford.smi.protege.owl.model.impl.DefaultOWLIndividual
>).

```

As a consequence the value of the individual has to be derived with a lookup function that retrieves the object.

**Do some check for constraints with functions** declared as

```

(deffunction ..)

```

These checks are typically to verify if the user has entered information that is contradictory or cannot lead to a solution. As already outlined the function CheckIfDimensionCompatible was written for this purpose.

**Retrieve variable values from facts.** This is done by retrieving the values stored in the properties of the UserQuestion fact-object and assigning the variable ?o\_qc to the con-

tainsSQueryCapability property:

```
(object (is-a UserQuestion)(containsSQueryCapability ?o_qc)
```

**Get the value for 'Query Capability'.** Now we have the value for the query capability we need the total query capability object to retrieve other values from it. In this example the variable ?qc is assigned to the query capability object belonging to the value that the user has entered.

```
?qc <-(object (is-a QueryCapability) (OBJECT ?o_qc) )
```

**Lookup the Data Model object belonging to the query capability and store it in the YourSolution individual.** The variable ?dm is assigned to the data model object supporting to the query capability value that the user has entered. The :NAME property of the data model object is retrieved and its value is stored in the hasDataModel property of the YourSolution individual.

```
?dm <-(object (is-a DataModel) (supportsSQueryCapability ?o_qc))  
(slot-insert$ YourSolution hasDataModel 1 (slot-get ?dm :NAME))
```

**Lookup the DBMS that support the Data Model.** The variable ?dm is used to determine the DBMS that supports the data model, and of the retrieved DBMS object the :NAME property is entered in the YourSolution individual.

```
(object (is-a Solution)(hasDataModel ?dm))  
?dbms <- (object (is-a DBMS) (supportsSDataModel ?dm))  
(slot-insert$ YourSolution hasDBMS 1 (slot-get ?dbms :NAME))
```

**Lookup values for other solution class values belonging to the DBMS.** This is now displayed as the code representing a Jess rule.

```
(defrule assign-DBMS "find DBMS belonging to DataModel"  
(object (is-a Solution)(hasDataModel ?dm))  
?dbms<- (object (is-a DBMS) (supportsSDataModel ?dm))  
=>  
(slot-insert$ YourSolution hasDBMS 1 (slot-get ?dbms :NAME))  
(slot-insert$ YourSolution hasPrice 1 (slot-get ?dbms hasPrice))  
(slot-insert$ YourSolution hasUserCommunity 1 slot-get ?dbms  
hasUserCommunity))  
)
```

As we can see, the values are inserted in a slot (property) and this result is as such displayed in the Protégé form of the YourSolution individual (earlier shown in Figure 29).

**If possible, generate DDL code.** There are three versions implemented in the prototype, one for PostgreSQL, one for PostGIS (displayed here) and one for Oracle Spatial. They

differ only in DDL declaration syntax, this is product specific. If the DBMS result is not either of these three, no DDL is generated.

```
(defrule generate-DDL-Postgis "create DDL for PostGIS"
(object (is-a UserQuestion)(containsSFDimension ?o_dim))
(object (is-a Solution)(hasDBMS ?dbms))
(object (is-a DBMS) (OBJECT ?dbms) (:NAME "PostGIS")))
=>
(printout t "CREATE TABLE my-table ( sensor-id INTEGER, mytime
DATE, sensor-value REAL);" )
(printout t "SELECT AddGeometryColumn('my-table', 'mylocation',
128, 'POINT', " ?o_dim " );")
(printout t "CREATE INDEX myindex ON mytable USING GIST (
mylocation );")
(stoploop 1)
)
```

This DDL code is displayed in the Jess console in JessTab.

Note that we have used the value of the variable that stores the dimension (?o\_dim) in the DDL code. This value was entered by the user when he specified how he wants to query in the user question form.

As can be seen a 'stoploop' function was necessary to prevent the code to write the DDL endlessly in the JessTab console. This happens because the fact remains true.

The code of the stoploop function is:

```
(deffunction stoploop (?x)
;
(while (<= ?x 1) do
break
?x<-(+ ?x 1))
break
)
```

ad c).

A ranking mechanism can be created by using certainty values as described in paragraph 3.2.2. This is not included in the prototype for reasons of time constraint, but the following is envisioned. Numerical data type properties are created and associated with solutions and requirements. For example: solution A contributes positively with a factor X to requirement B. As such the best solution for the requirement can programmatically be calculated. These solutions can be shown in the Protégé interface (the best solution listed above the next best) or by printing the ranking in the JessTab console.

ad d).

In some situations it is not possible to advise a solution because the data set does not contain necessary information (the data set does not contain space or time values in the

file). In these events an if-function in the prototype advises the user to perform some data pre-processing.

```
; check if data set has space attributes
(if (and (neq ?o_sv "space by coordinates in record") (neq (slot-
get ?o_qc :NAME) "qc-only-attributes"))) then
(printout t "Pre-processing is needed to obtain space and time
values in records" crlf)
break
else
return
)
```

The prototype experiment has proven that it is possible to create a system that can automatically advise solutions to requirements and that DDL code can be generated with values entered in a form during the system specification process.

To build a fully functional system the JessTab software contains however some serious limitations. These are the inability of JessTab to retrieve multiple values from a multivalue property and the absence of a relational database engine.

## 7.1 Property multivalues

First of all, it is not possible in JessTab to lookup all the values that belong to a multivalue property. In a property, more than one entry can be stored. As outlined already above, the reference to an individual is made via a pointer to the object (*<Java-Object:edu.stanford.smi.protegex.owl.model.impl.DefaultOWLIndividual>*). When multiple values are stored, they all contain the same pointer value as can be seen in Figure 32. This figure shows how three values are stored in the *supportsSQueryCapability* property (selected blue).

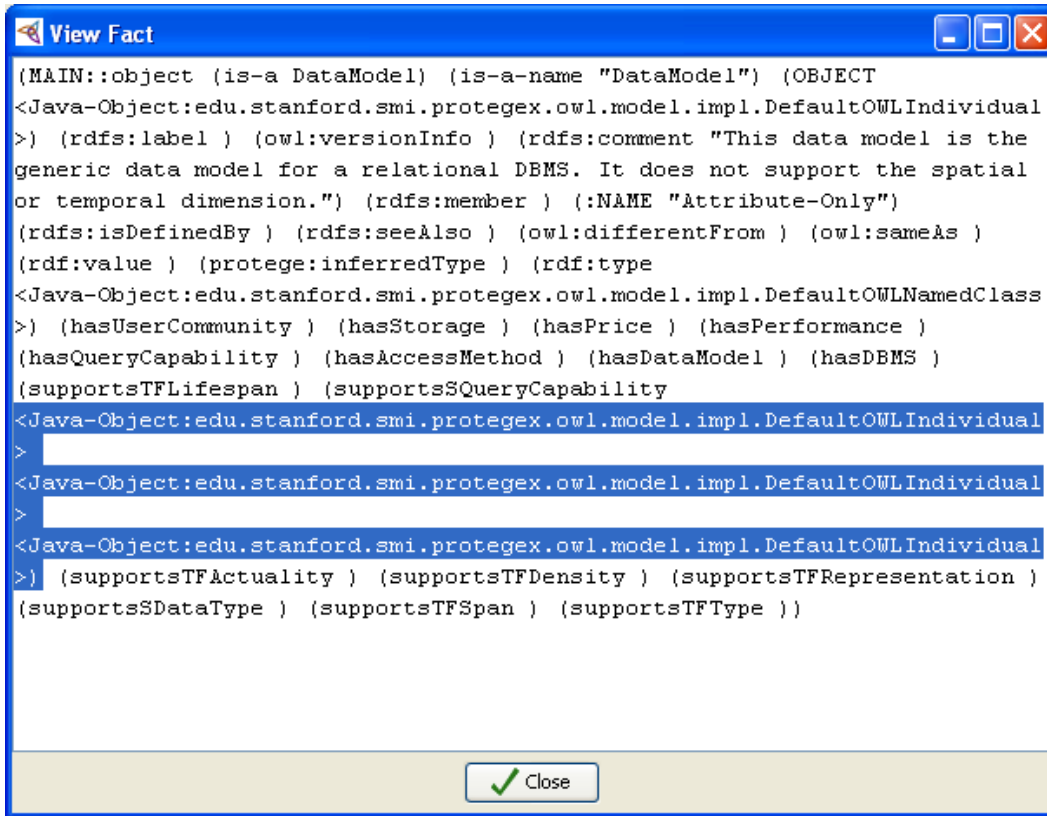


Figure 32: Multiple values stored in the property supportsSQueryCapability

JessTab cannot retrieve the corresponding individual values that belong to these values.

Executing the rule:

```
(defrule test-retrieve-multi-value
(object (is-a DataModel)(supportsSQueryCapability ?a)(:NAME
"S_Vector_ST_Geometry"))
?qc <-(object (is-a QueryCapability) (OBJECT ?a) )
=>
(printout t (slot-get ?qc :NAME) crlf))
```

with more than one value in the QueryCapability field would result in answer '0', meaning that 0 rules were executed (the LHS being FALSE). As a consequence 1-n and n-m relations are not supported.

Indication where the problem is caused is given by the JessTab error message '*bad index..*' that appears in the JessTab console when a multivalued variable is specifically addressed.

This limitation has been documented in [36], however not totally correct. It reads (page 43): '*property hasCommunication has to be restricted with single value because Jess rules can't detect more than one value and read them all.*' This is not correct, it should read: JessTab.. .

The reason why this is the case is probably explained by the JessTab author in [28], slide 67, where a reference to the index function of Jess is made indicating that there were *'implementation complexities'* because *'the Jess indexing scheme changes often (due to code optimizations)'*.

The above limitation has as consequence that only single values can be retrieved in property fields. This necessitates the creation of a very unrealistic ontology where only single property values are stored (for example a DBMS supporting only one query capability). A solution would be to write all possible combinations as individuals. This has been demonstrated by creating in the JessTab demo version of the ontology two entries for SDO geometry data type, one for the simple spatial query and one for the relation space query capability. This solution can be disqualified because it generates a lot of redundant information. Possible solutions for this problem are to modify the source code<sup>14</sup> for JessTab (it is open source) or to use a different rule engine that does support multivalued.

## 7.2 OWL data structure

The JessTab code is based on retrieving and manipulating values that are stored as facts. The facts are created in JessTab from the Protégé individuals with the *(mapclass ..)* command. The result is a list of facts that we can compare with rows in a database.

---

<sup>14</sup> [http://sourceforge.net/svn/?group\\_id=28307](http://sourceforge.net/svn/?group_id=28307)

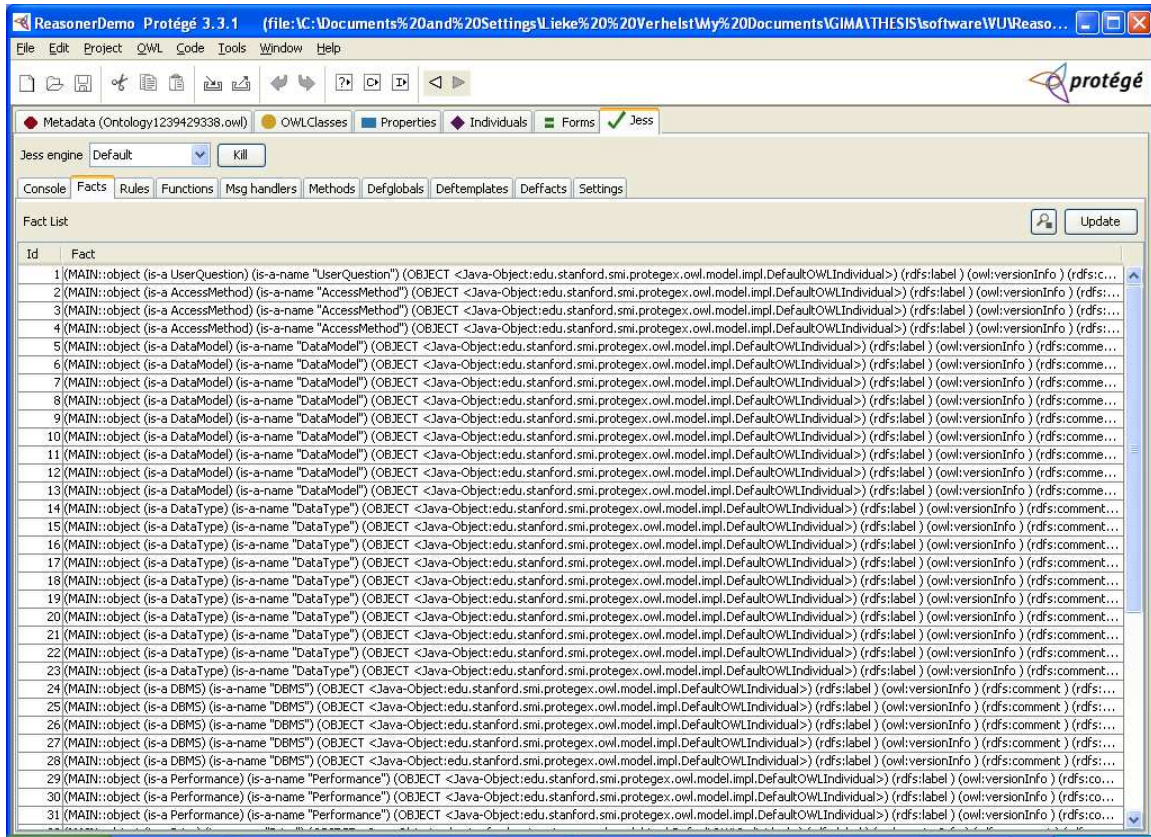


Figure 33: The data structure of the facts

Essentially we can compare the structure of the fact information with tables, columns and records: the classes are tables, the properties are columns and the individuals are records. The retrieval of the information via rules could be compared with querying the tables of a database.

If this was a relational database we would assign column unique ids and create tables that store relations between unique ids. This enables us to retrieve related information by executing SQL join queries. It is the relational database engine that takes care of this. How is this done in the ontology? In the ontology there are unique ids (the class names) and we have created relations when we selected individuals from other classes as values for the property fields. The following example compares the relational database join with an ontology 'join'.

We have two classes (tables), Class1 with properties value1 and value2, and Class2 with properties value3 and value4. We would like to find individuals of Class2 that have the same value in value3 as Class2 has in value2.

In SQL we would do this with one JOIN command, for example:

```
SELECT * FROM Class1 INNER JOIN Class2 ON Class1.value2 =
Class2.value3;
```

A substitute for the SQL join statement for joining Class1 and Class2 on a primary key (value2) -foreign key (value3) in a Jess rule would be:

```
(mapclass Class1)
(mapclass Class2)
(mapclass JoinClass)

(defrule makejoin
(object (is-a Class1)(value1 ?x)(value2 ?y))
(object (is-a Class2)(value3 ?y)(value4 ?z))
=>
(slot-insert$ MyJoinClass joinvalue1 1 ?x)
(slot-insert$ MyJoinClass joinvalue4 1 ?z)
)
```

This Jess code fills the properties joinvalue1 and joinvalue4 of the MyJoinClass with the values value1 from Class1 and value4 from Class2 whenever value2 equals to value2. To obtain a new *set* of individuals MyJoinClass of class JoinClass this code has to be looped. In every loop a new individual MyJoinClass has to be created. Since every class name has to be unique the MyJoinClass name has to be changed every time, for instance with a sequence number. The loop must end when no new values of Class2 that have the same value3 as value2 are found.

In addition to this Jess needs to know beforehand how many properties there will be in the new individual. A solution where the number of properties is unknown (such as 'select \* from ..' in relational database technology is not possible.

To summarize: with the Jess rule code, the lookups have to be programmed and the number of properties have to be known, while when using a relational database one SQL join statement will make the database engine retrieve all the values for you, even when the number of column values is unknown.

So an OWL file is, however relational in its structure, in its living environment not supported by a relational (database) engine. How *data* is linked in the open world of RDF (OWL) and what problems this gives is described in [14]. The principle of linking data is the same as with HTML: use URI's as names for things and use the HTTP protocol to transport lookup requests. To use URI's is common in HTML, but this is not so widely used with data. Berners-Lee sees this as something temporary caused by the fact that this is new technology. To browse data a SPARQL server is recommended in [14]. SPARQL is based on querying triplets. It uses variables in a query to refer to the corresponding parts of a triple. How this can help overcome the issues addressed here is subject for further research.

This chapter has outlined how a rule engine can be used to work with information stored in an ontology. It has also outlined some limitations from ontologies in general and from JessTab in particular. The next chapter will show an alternative manner to work with knowledge stored in an ontology, namely the use of a reasoner. It will address how a reasoner works on the prototype and it will outline the limitations of this method.

## 8 Using the reasoner

In this chapter it is described how the knowledge that is stored in an ontology can be accessed and queried using a reasoner. Also the limitations of this method are addressed with regard to the thesis objective.

The work in this chapter is complemented with the 'ReasonerDemo' prototype. The ReasonerDemo ontology OWL file is listed in appendix 3.

The limitations caused by JessTab demand the investigation of an alternative solution. One alternative is making use of a reasoner. The concept is already described in paragraph 3.3.4.

The task to be completed is to make the reasoner select solutions satisfying requirements. This can be done by defining rules in Description Logic (DL) and enter these in the Protégé editor. How this can be achieved is described in [20] and summarised below:

### **Define empty classes and give them a meaningful name**

Three example classes have been created to experiment with this: My\_Sol-NoPrice, My-Sol-DM-Attribute-Only-no-price and My\_Sol-QC-space-relation.

Example 1: The Solution sub-class 'My\_Sol-NoPrice' was created to represent individuals that have the value 'no-price' in property 'hasPrice'.

Example 2: The Solution sub-class 'My-Sol-DM-Attribute-Only-no-price' was created. This class represents individuals of data models that have value 'Attribute Only' for property 'supportsSDDataModel'. It also represents individuals that have the value 'no-price' in property 'hasPrice'.

Example 3: A Solution sub-class 'My\_Sol-QC-space-relation' was created to represent individuals that have the value 'space-relation' in property 'supportsSQueryCapability'.

### **Assign DL rules to these classes**

In principle there is no limitation to what DL rules can be assigned to the classes as long as the rules are made of valid elements (classes, properties, individuals, values) of the ontology (they must be selected via the Protégé interface).

The result of a rule is either that an individual is inferred or it is not inferred. When an individual is inferred it shows up in the Inferred tab of the Protégé editor in the example class and the number of inferred individuals is shown next to the class name (see Figure 34, in this case there is one inferred individual). When nothing is inferred, the inferred tab of the example class remains empty and the number next to the class is (0/0).

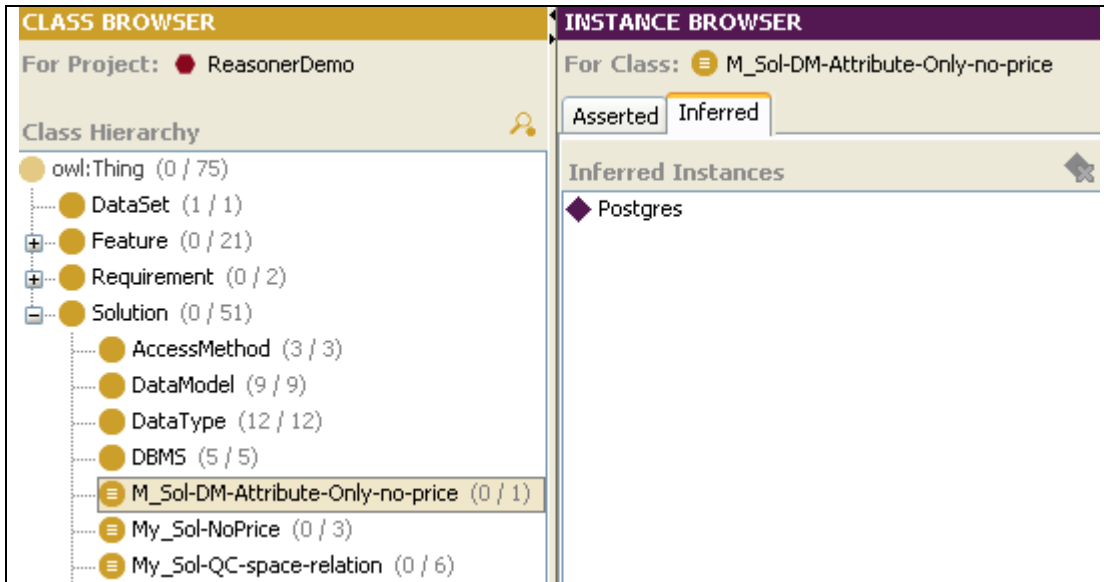


Figure 34: An inferred individual <sup>15</sup>

The rules that were assigned define a *necessary and sufficient* condition. This means that ‘the conditions are necessary for membership of the class and they are sufficient to determine that any other individual that satisfies the conditions must be a member of the class’. [20]

The rules for the three classes (notated here in the syntax as they are entered in Protégé which is a user interface simplification of DL) are respectively:

Example 1:

Solution property hasPrice hasValue ‘no-price’

Example 2:

Solution property supportsSDataModel hasValue ‘Attribute Only’ AND Solution property ‘hasPrice’ hasValue ‘no-price’.

Example 3:

Solution property ‘supportsSQueryCapability’ hasValue ‘space-relation’.

---

<sup>15</sup> The DBMS name ‘Postgres’ in this screen is referred to in the text as PostgreSQL.

They are entered in the Protégé interface resulting in the following screen (Figure 35)

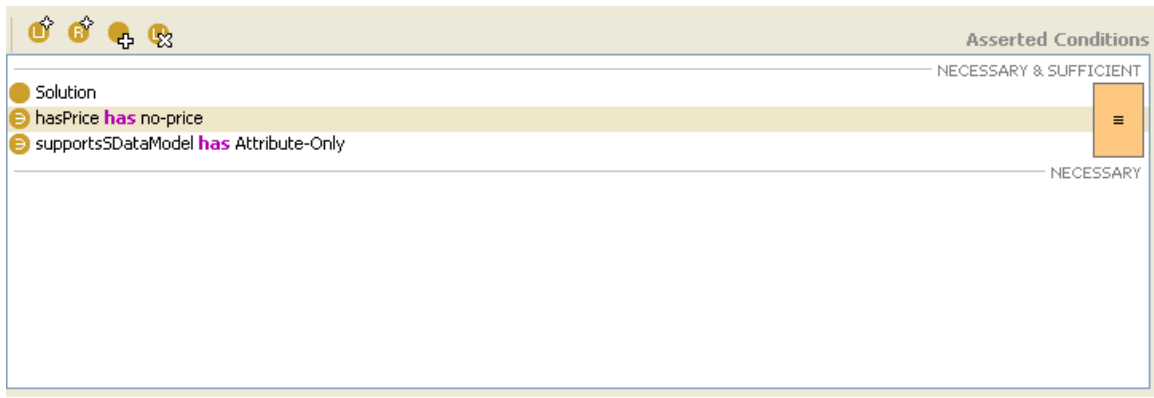


Figure 35: The rules entered in the Protégé editor (Example 2)

### Run the reasoner in the 'compute inferred types' mode

The Pellet reasoner must be started in a DOS prompt with the command 'pellet dig'. The RACER reasoner can just be started by executing the RacerPro executable. To run the reasoner in Protégé, the menu command 'OWL-Compute inferred types' must be given.

**Display the individuals that are listed as inferred individuals.** These are the individuals that comply with the DL rules that were assigned. The result of this is shown in the figure below (Figure 36, which is an enlargement of Figure 34).

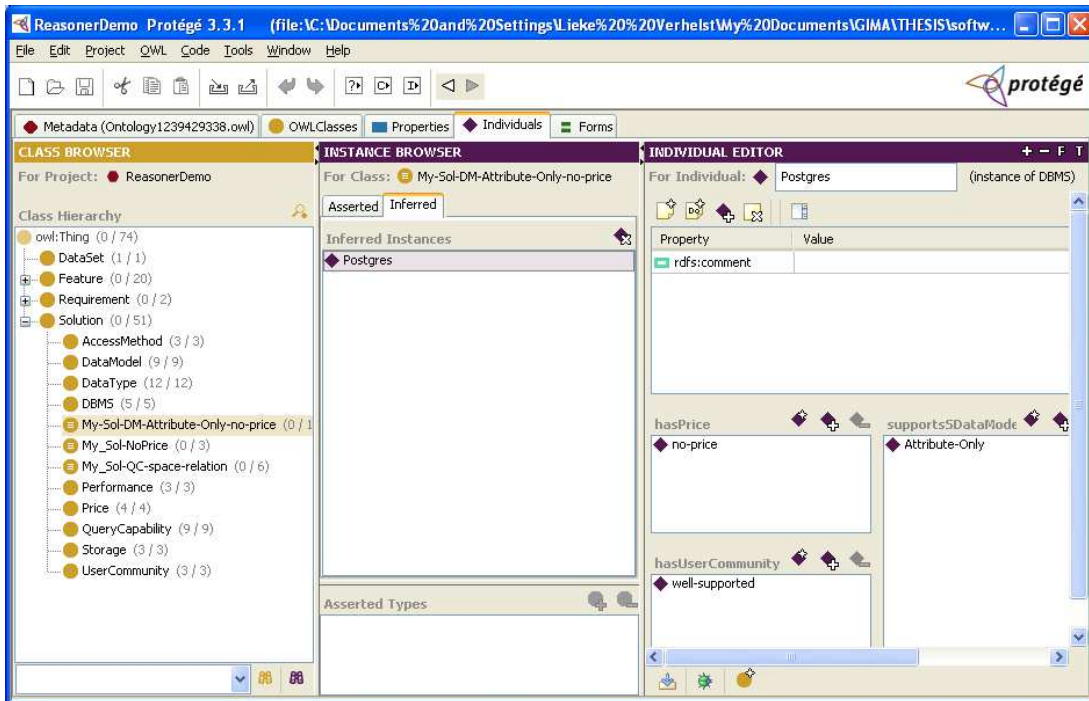


Figure 36: The result of inferring individuals with a reasoner

In this example the reasoner has given the solution DBMS=PostgreSQL for the user requirement that the DBMS must be capable of querying 'attribute only' and have as price value 'no price'.

The result of the reasoning is in fact the classification of individuals into the created example 'container' class (My-Sol\_DM\_Attribute-Only-no-price) that has been predefined. The individuals comply to the DL rules or not. As a consequence only **existing** individuals that have the requested value stored in one of their properties are inferred. The reasoner cannot create **new** individuals with properties from various individuals like in the JessTab example the YourSolution class. Therefore one has to know beforehand what property information is needed as solution values in the inferred individual. This must be specified when the ontology is created, because then properties and values are assigned to classes.

## Ranking

When more than one individual complies with the DL statements listed in the newly created class we would like to know if they can be ranked. This can be accomplished by writing Description Logic rules to be evaluated by a reasoner. A proposal for this method can be found in [37] where a case study of buying a computer with certain capabilities is discussed. In this work ranking is based on how many properties of the user requirements have been satisfied by the inferred individual. This determines the ranking. The decision process for buying is similar to the process of selecting the best

solution for the requirements, therefore the in [37] proposed algorithm is a candidate for implementation in future research.

This chapter has outlined the method for querying an ontology by means of a reasoner. It has shown the limitations of this method with regard to the thesis objective.

## 9 Comparing the solutions

This chapter describes how the JessTab solution and the solution of the reasoner score on the predefined instrument requirements. Also some comparison with an imaginary solution with UML is given.

How the two prototypes score on the requirements listed in 1.2.3 is given in Table 4 and discussed below.

**Table 4: The reasoner solution versus the JessTab solution**

Requirement	OWL ontology with reasoner	OWL ontology with rule engine (JessTab)
1. storing and retrieving existing knowledge	++	-
2. user must be able to enter specification information	--	++
3. flexible enough to adapt new knowledge	++	-
4. automatically guide the user	+	+/-
5. provide a ranking order	n/a	n/a
6. generate code	--	++

As for requirement 1, an ontology is meant for storing and retrieving knowledge, this qualified as ++ in the reasoner solution. When JessTab is used some knowledge must be removed from multiple value properties to make the rules run so this affects negatively the score of the JessTab solution to a -.

Requirement 2: As for entering solution selection criteria as DL to be used by a reasoner, this is really not something one can expect from an end-user, so --. The JessTab solution allows the user to interactively enter user requirements via a form. This was the intention of the instrument so a ++ was given.

Requirement 3 must be compared with requirement 1. Specifically here is meant how easy it is to add new existing knowledge to the instrument. As already outlined in requirement 1, an ontology is meant for interoperability with other ontologies, so ++. And JessTab does not work with multiple values, so you *can* add new information only when you do it single value properties. A - is scored.

As for requirement 4: the programmability of the solutions. With JessTab you can program whatever you like however this is not as flexible as when the underlying data was supported by a database engine so a +. The reasoner gets a '+/-' since only existing

individuals can be classified and no new combinations of individual properties can be created.

Since requirement 5 has only been envisioned and not implemented we cannot compare.

Requirement 6: a reasoner cannot generate code, so this solution gets a - - for requirement 6. The JessTab solution can generate code and it can even use values that were entered by the user so a + +.

We can conclude that neither of the two solutions could satisfy sufficiently all the requirements that were posed at the beginning of the research. The question that rises probably now is: how would UML compare? OWL and UML are developed for very different purposes but we can perhaps phrase the comparison differently. If we were to build the instrument again and now with UML, how would the outcome be?

We would define UML classes similarly like we did in OWL. We would create data tables from the classes via a model transformation (class diagram to DDL) in a relational database and populate them with values of existing phenomena. 1-n and n-m relations between the tables are supported (in contrast with the JessTab solution). We would create an application with a form that the user can use to enter requirements and information about the data set. With underlying SQL queries we would have the relational database find corresponding values of solutions which we then present to the user in some kind of user interface. This solution would typically score much better on requirements 4 and 6. However the application built would not easily support adaptation of new knowledge for which new tables have to be created. This solution would score less on requirement 3.

It is important to realise what was really built in this research. We built an expert system that can generate DDL code. It is an expert system because it advises by means of rules that run on stored knowledge. It *selects* a database model by criteria and presents it *by name* and criteria. It does not *create* a database model, as UML would typically do, meaning machine understandable and DBMS specific descriptions of tables, columns, indexes. To obtain this functionality, the prototype could be extended to contain real database models (as machine understandable descriptions) and not just the name and characteristics of them. This has not been done in this research because the machine understandable descriptions of the database models used in this research live in prototypes on computer systems of academic institutes and can probably only be used in that environment.

To conclude this chapter, based on the experiences of the research, a short recommendation is given for in which environments to use OWL or UML:

- use UML to design complex technical implementations such as OO programs with underlying databases
- use UML when structures are not frequently changing

- use OWL in open and distributed environments like the Internet
- use OWL to store and retrieve existing knowledge of concepts and their relations

To build the instrument as was intended at the beginning of the research we would probably benefit from both UML and OWL. UML for the design of the instrument itself and OWL for the underlying knowledge base mechanism. It is a challenge however to combine static and dynamic requirements.

To use UML next to OWL has also been envisioned by the OMG by specifying mapping mechanisms from one to the other in the ODM [23].

This chapter has compared the two possible methods to access and query the prototype. It has shown how the two methods score on the prototype requirements that were set in paragraph 1.2.3. It has also addressed if and how UML could have been used in the design of the instrument.

# 10 Conclusions and recommendations

In this research the objective was to provide users of sensor data with an instrument that advised them on how to design a database in such a way that it can optimally answer their questions about the sensor data. At the very early stage there was no clear picture of what this instrument would eventually turn out to be. Ideas ranged from flowcharts to a computer program with many IF..THEN.. statements. An extensive literature study accompanied by brainstorm sessions with supervisors and 'invited guests' (Rob Lemmens and Marian de Vries) was the right mix that produced the eventual end result.

## 10.1 Summary and discussion

The work of [8] and [5] was a great help during the literature research on spatio-temporal modelling. These works contain an overview of academic spatio-temporal research over the last two decades. In addition a clear description is given of how spatial and temporal semantics relate to data models. Because of this it has been used as the main source of information for the research on spatio-temporal semantics and database modelling.

How to find the right approach to design the instrument took a far more effort. In fact it has been difficult not to drown in new and interesting study areas. Fortunately guidelines were found by researching solutions for system requirement specification in literature. The eventual approach of designing an ontology and using a reasoner and rule-engine to work with the knowledge stored in the ontology is based on the work by Kroha and Gayo [26]. This provided the basis for the approach as well as the tools used. Steps describing how the 'trick could be done' were found in [36].

The creation of the ontology was the most important and challenging part of the research. First of all the determination of *what* needed to be stored in the ontology was important and next *how* this must to be stored.

Early in this process it became apparent that, thinking in requirements and solutions, four main elements are driving the design of a spatio-temporal database. These are the features of time, space and time-space, the user requirements (the user question and the non-functional requirements), the existing technical solutions (such as data models, data types, DBMS, access methods) and the properties of the data set (size, update frequency, which data is available in the data set, dimension, geometry). This information then must be stored in the ontology. But how? It appeared that little directions can be given beforehand for how an ontology should be built. A basic framework of how the most important classes related to each other was designed. Next it had to be determined how the existing knowledge must be entered and how relations between classes and properties should be defined. How this was dealt with is described in chapter 6.

As an instrument to work with the knowledge stored in the ontology the rule engine JessTab and a reasoner were chosen. The exercise with JessTab showed that values from the user specification process can be used to provide a solution, being (a combination of) individual values or a textual response on the user interface. However some important limitations ceased further prototype development. These limitations were the inability of JessTab to retrieve multivalued property values and the absence of a relational engine which made querying the ontology data limited, complex and tedious. This was described in chapter 7. An advantage of JessTab was that it could generate DDL code, even containing variables that were retrieved from the user specification input.

Working with the reasoner learned that interesting results are easily generated. However the results were nothing more than a classification of existing individuals. No new individuals with combinations of properties from other individuals could be created (as was the case with JessTab). Therefore the reasoner results greatly depend on a) the structure of the ontology and b) the understanding of Description Logics of a user. It is not feasible to ask a user of the prototype instrument to enter DL code in order to generate the required advice. This was described in chapter 8.

A comparison between the JessTab solution and the reasoner solution was given in chapter 9. Having them side by side showed that neither solution scored sufficiently on the requirements defined at the beginning of the research. The reasoner solution scored higher on the flexible storage of knowledge than the JessTab solution because of the limitation of JessTab to support 1-n and n-m relations. JessTab scored higher on the user interface requirement for it provides a form user interface, whereas the reasoner interface requires the user to enter complex DL in the Protégé interface to obtain the solution.

How the instrument could have been built with UML is also envisioned in chapter 9. With UML it would have been easier to query and present the information to the user because a custom application would have been built. However this application needs to be rewritten when new tables for knowledge must be added, so this solution is not as flexible as an ontology based solution.

Before formulating main conclusions first the research sub-questions will be addressed.

## 10.2 Answers to sub-questions

Starting from scratch the beginning was a literature study to answer the following research sub-questions:

- what are the most important characteristics of spatio-temporal data?
- what are user questions and how can they be categorised?

- which technical solutions are available that take care of a proper handling of spatio-temporal data?

Information from the work of Pelekis c.s. was complemented with knowledge of relational data modelling, spatial data modelling and temporal data modelling as well as existing technical solutions. The semantics of time and space, caught in a data model, together with the query capabilities of a database system characterise the capabilities of the system. The sample frequency of the sensor determines important database performance factors like file size and data update frequency. The database capabilities for indexing and clustering deal with these factors, however to quantify and compare these capabilities is technically too complex and therefore kept out of scope of the prototype solution calculation. They are stored as knowledge in the ontology however.

The user questions can be categorised in 9 types:

attribute only, spatial-simple, spatial-relationships, temporal simple, temporal range, temporal relationships, spatio-temporal simple, range and behaviour.

It is mainly the data model that determines which questions the user can effectively pose on the database system. Not many spatio-database models are implemented as a working solution. However existing relational, spatial and temporal databases *can* offer solutions to store and query sensor data. It must be understood that their capabilities for executing complex spatio-temporal queries is then limited.

The next sub-questions relate to the techniques used to build the instrument.

- what W3C or OMG specification can we use creating the instrument?
- which tools can be used to design and create the instrument?

Literature research learned that techniques from both Artificial Intelligence and Software Engineering can be used for creating functions of the instrument to be designed. Moreover it was discovered that AI methods and MDA are recently integrated by the ODM standard. Several ODM specifications were evaluated against the instrument requirements in chapter 5. It was decided to use OWL because of its capabilities for storing existing knowledge and adapting new knowledge, and its possibilities to provide a real end-user interface and to work with (reason on) individuals.

- how can we model the relation between sensor data concepts and a suitable database model?

This is done by first storing the relation between the sensor data concepts and a data model in a table (Table 1). This table is then translated into the ontology. This was covered in chapter 6.

- how can we assign a weight qualification for (parts of) the model?

This can be achieved by assigning weight values to properties. This can be used both in the reasoner solution and the rule engine solution. In the reasoner solution it is with DL code that the weight values must be evaluated, in the rule engine solution this can be done with Jess code. Implementations of this were not provided in the thesis work because of time constraints.

### 10.3 Main conclusions and recommendations

In this research the capabilities of semantic modeling for designing a spatio-temporal database have been evaluated. In this process also the 'old' method UML has passed by to compare. It has been proved that for the intended purpose no single technology delivered sufficient results. However, it has been proved that a combination of semantic technologies and UML will deliver better results. This leads to the first of the main conclusions: **a modeling technique works best in the environment for which it was designed**. The ODM has provided a standard for using OWL next to UML.

The initial thought behind the research was that a proper specification of the database design helps to achieve better application results. It was the intention of the research to design an instrument that helps to overcome the issue of data modeling. The research has proven however that **the data structure of an ontology determines the extent to how it can be used**. This is the second main conclusion of the research.

Working with semantic technologies has learned that to obtain a valuable result it is necessary to select the right tools and execute a proper data analysis. This leads to the third main conclusion: **designing systems in open environments like the Internet requires a careful system design process**

The recommendation therefore is to use the same computer science 'common sense' methods when using semantic technologies as in other areas of computer engineering. These are: to conduct a proper tool selection process, to perform a data analysis and to use standards and software development methods.

## 10.4 Suggestions for further research

In this research just two of the many knowledge retrieval methods for ontologies have been evaluated. Their shortcomings have been described. They are in short: for rule engines there is no relational engine support, resulting in complex querying. For reasoners: the insufficient user interface. To overcome this additional research is proposed. Some have already been mentioned earlier, they will be repeated here.

To find better mechanism for querying ontologies *ontology query languages* can be researched. The one that has been recommended by the W3C and mentioned earlier in paragraph 7.2 is 'SPARQL Query Language for RDF' [38]. Another option is to research how the Oracle Semantic Technologies support querying [39] .

Also many other rule engines exist. Perhaps there is one that does not suffer from the lack of support for multivalued relations like JessTab. A starting point for further study would be the home page of the Rule Interchange Format (RIF) Working Group of the W3C<sup>16</sup>. Referenced to often is Semantic Web Rule Language (SWRL), a proposal from W3C [40]. An alternative to overcome the multivalued issue would be to modify the JessTab code.

Also interesting to mention here is the Object Constraint Language (OCL). This specification for UML [41] enables developers to query UML and specify operations similar to rules.

Another suggestion for further research is to manually program the generation of DDL code on top of the reasoner solution. The proposed idea is: infer with the reasoner individuals that carry information about the solution. Next, investigate if it is possible to retrieve via program code the property values of the inferred individuals. Use these to compose the DDL code for the to be generated tables, columns and indexes.

---

<sup>16</sup> [http://www.w3.org/2005/rules/wiki/RIF\\_Working\\_Group](http://www.w3.org/2005/rules/wiki/RIF_Working_Group)

# Bibliography

1. Botts, M., Percivall, G., Reed, C., Davidson, J., *Sensor Web Enablement: Overview And High Level Architecture*. 2007, OGC White paper.
2. Durity, S., *Introduction to the TimeSeries DataBlade*. 2005.
3. OMG. *Introduction to OMG's Unified Modeling Language*. 2005 [cited; Available from: [http://www.omg.org/gettingstarted/what\\_is\\_uml.htm](http://www.omg.org/gettingstarted/what_is_uml.htm)].
4. IBM, *IBM Informix TimeSeries DataBlade Module User's Guide*. 2001.
5. Pelekis, N., Theodoulidis, B., Kopanakis, I., Theodoridis, Y., *Literature review of spatio-temporal database models*. *The Knowledge Engineering Review*, 2004. **19**(3): p. 235–274.
6. Giannotti, F., Pedreschi, D., *Mobility, Data Mining and Privacy - Geographic Knowledge Discovery*. *Mobility, Data Mining and Privacy*, ed. F. Giannotti, Pedreschi, D. 2008: Springer.
7. Güting, R.H., Schneider, M., *Moving Objects Databases*. 2005: Morgan Kaufmann.
8. Pelekis, N., *STAU A Spatio-Temporal Extension for the Oracle DBMS*. 2002, University of Manchester.
9. Lemmen, C., *Product Survey on Geo-databases*. *GIM international*, 2007. **May 2007**.
10. Gasevic, D., Djuric, D., Devedzic, V., *Model Driven Architecture and Ontology Development*. 2006: Springer.
11. Mooney, R.J., *First Order Predicate Logic*, in *Artificial Intelligence*. 2009, University of Texas.
12. Berners-Lee, T., *Semantic Web - XML2000*. 2000, W3C Talk.
13. W3C, *RDF Primer*, F. Manola, Miller, E., Editor. 2004, W3C Recommendation.

14. Berners-Lee, T. *Linked Data (Status: personal view only. Editing status: imperfect but published. )*. <http://www.w3.org/DesignIssues/LinkedData.html> 2007 [cited.
15. McGuinness, D., Harmelen, F van. , *OWL Web Ontology Language Overview* 2004.
16. Smith, M.K., Welty, C., McGuinness, D. L., *OWL Web Ontology Language Guide*. 2004.
17. Lemmens, R., *Semantic Interoperability of distributed Geo-Services*. 2006, Delft University of Technology.
18. Nardi, D., Brachman, R.J., *An Introduction to Description Logics*, in *the Description Logic Handbook*, F. Baader, Calvanese, D. , McGuinness, D.L., Nardi, D. , Patel-Schneider P.F., Editor. 2002, Cambridge University Press. p. 5-44.
19. Donini, F., Lenzerini, M., Nardi, D. , Schaerf, A., *Reasoning in Description Logics*, in *Principles of Knowledge Representation and Reasoning*, G. Brewka, Editor. 1996, CLSI Publications. p. 193-238.
20. Horridge, M., Knublauch, H., Rector, A., Stevens, R., Wroe, C., *A Practical Guide To Building OWL Ontologies Using The Protege-OWL Plugin and CO-ODE Tools Edition 1.0*. 2004.
21. Friedman-Hill, E., *Jess in action*. 2003: Manning.
22. OMG, *MDA Guide Version 1.0.1*, J.M. Miller, J., Editor. 2003.
23. OMG, *Ontology Definition Metamodel, version 1.0*. 2009.
24. Ambler, S.W. *UML 2 Use Case Diagrams*. Agile Modelling 2003 [cited 2009 June 12]; Available from: <http://www.agilemodeling.com/artifacts/useCaseDiagram.htm>.
25. Xu, W., Dilo, A., Zlatanova, S., van Oosterom, P. *Modelling emergency response processes: Comparative study on OWL and UML*. 2008.
26. Kroha, P., Gayo, J.E.L., *Using Semantic Web Technology in Requirements Specifications*. 2008.

27. Savely, R., *CLIPS reference manual, Volume I -Basic Programming Guide, Quicksilver Beta*. 2007.
28. Eriksson, H., *JessTab Tutorial*. 2006, powerpoint presentation.
29. PostgreSQLGlobalDevelopmentGroup, *PostgreSQL 8.3.7 Documentation*. 2008.
30. Kothuri, R.G., A. , Beinat, E., *Pro Oracle Spatial for Oracle Database 11g*. 2007: Apress.
31. Neufeld, K., *PostGIS 1.3.5 Manual*. 2008.
32. Noy, N., McGuinness, D.L., *Ontology Development 101: A Guide to Creating Your First Ontology*. 2001, Stanford University.
33. NASA, *An ontology of time*, in <http://sweet.jpl.nasa.gov/1.1/time.owl>.
34. W3C, *An ontology of time*, in <http://www.w3.org/2006/time>.
35. Eriksson, H., *JessTab Manual, Integration of Protégé and Jess*. 2004, University of Linköping.
36. Sun, B., *Modelling of Interaction Units*. 2005, University of Linköping.
37. Fan, Z., Chen, X., *A Ranking Algorithm Based on Service Capability in Semantic Web Service Discovery*. 2008.
38. Prud'hommeaux, E. and A. Seaborne, *SPARQL Query Language for RDF*. 2008.
39. Oracle, *Oracle Database Semantic Technologies Developer's Guide*. 2008.
40. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosz, B., Dean, M., *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. 2004, W3C Member Submission.
41. OMG, *UML 2.0 OCL Specification*. 2006.



# Appendix 1 Example Ontologies

Some locations where one can find ontologies related to geo-information, time and space are:

<http://swoogle.umbc.edu/>

<http://www.ontologyportal.org/>

<http://www.planetont.org/share/>

<http://www.geospatialmeaning.eu/category/geo-ontologies/>

<http://www.ordnancesurvey.co.uk/oswebsite/ontology/>

<http://protege.stanford.edu/download/ontologies.html>

<http://sweet.jpl.nasa.gov/1.1/>



# Appendix 2 JessTab

## Installation

JessTab is provided with the base install of Protégé. It needs the Jess jar file to function. The Jess.jar can be downloaded from <http://www.jessrules.com/jess/download.shtml>. To install, copy only the Jess.jar file to the JessTab plugin directory of Protégé. In windows this is typically 'C:\Program Files\Protege\_3.3.1\plugins\se.liu.ida.JessTab\'

## To run the demo:

Check the values stored in the YourSolution individual (they should be empty)

Select your values in the DataSet class and the UserQuestion class

Type '(batch <path-name-to-jesscode-run.txt>)' in the Jess console (use '\\\' as path separator)

Type '(run)', the rules are now executed on the facts.

To end the code use the Break button in the Jess console interface

## Jess Code

```
;clear old facts and rules

(clear)

; load Protege instances as Facts

(mapclass Feature)
(mapclass Requirement)
(mapclass DataSet)
(mapclass Solution)

; check if data set dimension is smaller than user question
;dimension

(deffunction CheckIfDimensionCompatible (?x ?y)
  (if (< ?x ?y) then
    (printout t "The dimension of the dataset is smaller than the
requirements. This is not possible." crlf)
    break
  else
    return
  ))

;prevent looping

(deffunction stoploop (?x)
  (while (<= ?x 1) do
    break
    ?x<-(+ ?x 1))
  break
  )
```

```

; obtain necessary values from facts
(defrule collect "collect all user information and assign data
model"
(object (is-a NFR)(hasPerformance ?o_perf)(hasPrice
?o_pr)(hasStorage ?o_stor) (hasUserCommunity ?o_uc))
(object (is-a UserQuestion)(containsSQueryCapability
?o_qc)(containsSFDimension ?o_dim)(containsTFActuality
?o_act)(containsSFGeometry ?o_geom))
(object (is-a DataSet)(ds-attribute-value ?o_av)(ds-time-value
?o_tv)(ds-space-value ?o_sv)(containsSFGeometry
?o_dsgeom)(containsSFDimension ?o_dsdim)
(containsTFActuality ?o_dsact))
=>
(CheckIfDimensionCompatible ?o_dsdim ?o_dim)

; check if data set has space attributes

(if (and (neq ?o_sv "space by coordinates in record") (neq (slot-
get ?o_qc :NAME) "qc-only-attributes"))) then
(printout t "Preprocessing is needed to obtain space and time
values in records" crlf)
break
else
return
)

(printout t "all data collected." crlf)
)

; lookup DataModel related to Query Capability

(defrule lookup-data-model "lookup data model related to QC"
(object (is-a UserQuestion)(containsSQueryCapability ?o_qc))
?qc <-(object (is-a QueryCapability) (OBJECT ?o_qc) )
?dm<- (object (is-a DataModel) (supportsSQueryCapability ?o_qc))
=>
(printout t "The user entered query capability is: " (slot-get ?qc
:NAME) ". " crlf)
(printout t "The Data Model supporting this requirement is: "
(slot-get ?dm :NAME) ". " crlf)
(slot-insert$ YourSolution hasDataModel 1 (slot-get ?dm :NAME))
)

; renew mapping of Protege Solution instance to Jess

(mapclass Solution)

; lookup DBMS related to DataModel and store result DBMS
;capabilities in Solution instance

(defrule assign-DBMS "find DBMS belonging to DataModel"
(object (is-a Solution)(hasDataModel ?dm))
?dbms<- (object (is-a DBMS) (supportsSDataModel ?dm))
=>
(slot-insert$ YourSolution hasDBMS 1 (slot-get ?dbms :NAME))
(slot-insert$ YourSolution hasPrice 1 (slot-get ?dbms hasPrice))
)

```

```

(slot-insert$ YourSolution hasUserCommunity 1 (slot-get ?dbms
hasUserCommunity))
)

(mapclass Solution)

; when possible display DDL

(defrule generate-DDL-PostgreSQL "create DDL for PostgreSQL"
  ?a<-(object (is-a Solution)(hasDBMS ?dbms))
  (object (is-a DBMS) (OBJECT ?dbms) (:NAME "PostgreSQL"))
  =>
  (printout t "CREATE TABLE my_table ( sensor-id integer,
mylocation varchar, mytime time, sensor-value integer);" )
  (printout t "CREATE INDEX my_index ON mytable (sensor-id);" )
  (stoploop 1)
)

(defrule generate-DDL-Oracle "create DDL for Oracle Spatial"
  (object (is-a Solution)(hasDBMS ?dbms))
  (object (is-a DBMS) (OBJECT ?dbms) (:NAME "Oracle-Spatial"))
  =>
  (printout t "CREATE TABLE my_table ( sensor-id NUMBER, mylocation
SDO_GEOMETRY, mytime DATE, sensor-value NUMBER);" )
  (printout t "CREATE INDEX my_index ON my_table (mylocation)
INDEXTYPE IS MDSYS.SPATIAL_INDEX;" )
  (stoploop 1)
)

(defrule generate-DDL-Postgis "create DDL for PostGIS"
  (object (is-a UserQuestion)(containsSFDimension ?o_dim))
  (object (is-a Solution)(hasDBMS ?dbms))
  (object (is-a DBMS) (OBJECT ?dbms) (:NAME "PostGIS"))
  =>
  (printout t "CREATE TABLE my-table ( sensor-id INTEGER, mytime
DATE, sensor-value REAL);" )
  (printout t "SELECT AddGeometryColumn('my-table', 'mylocation',
128, 'POINT', " ?o_dim " );")
  (printout t "CREATE INDEX myindex ON mytable USING GIST (
mylocation );")
  (stoploop 1)
)

```



# Appendix 3 The OWL Ontology

The following pages display the OWL code of the JessTabDemo ontology and the ReasonerDemo ontology. This code is generated by the Protégé editor from the input that was entered in the editor.

The code is listed here for those who want to view the ontologies or work with the JessTabDemo or ReasonerDemo. The code listed on the next pages must then be copied into a text editor and saved as a file with extension .owl. The thus obtained file must be loaded in an ontology editor such as Protégé.

To run the ReasonerDemo:

- load the ReasonerDemo.owl file in Protégé
- start a reasoner such as Pellet or RACER
- in the OWL menu of Protégé select 'compute inferred types'
- the reasoner will run and the inferred types are shown on the inferred tab in the individuals browser.

To run the JessTabDemo:

- load the JessTabDemo.owl file in Protégé
- load the Jess code in the Jess console
- run the Jess code by executing the '(run)' command in the Jess console