WARSAW UNIVERSITY OF TECHNOLOGY

FACULTY OF MATHEMATICS
AND INFORMATION SCIENCE

MASTER THESIS
COMPUTER SCIENCE

Intelligent Techniques Applied to Management of Ontologically Demarcated Information Gathered from the Internet

Inteligentne techniki zarządzania ontologicznie opisaną informacją pozyskiwaną z Internetu

Author: Paweł Olesiuk

Supervisor: Dr. Marcin Paprzycki

WARSAW 2008

Summary

The objective of this thesis is to develop the Content Management Subsystem (CMS) that is a part of the Travel Support System (TSS) – the academic agent-based project that has come into being to check and verify the power of the Semantic Web and Multi-agent Systems. The CMS is mainly responsible for managing the travel-related data in the storage of the Travel Support System – data in the storage has to be up to date, reliable and complete. TSS is the system for supporting travellers' needs. In the scope of travelling the design of the TSS includes such aspects as: standard transportation, choices of accommodation, restaurants, movie theatres, national parks, historical sites and other points of interest. The data in the TSS is ontologically demarcated and all aspects of managing data are performed by software agents.

The designed and implemented system includes several agents that form a Multi-agent System and that co-operate which each other to meet requirements related to functionalities of the CMS. Agents are designed according to the *decomposition* methodology and are implemented in JADE (Java Agent DEvelopment framework). Two additional ontology models were defined: *OntologySource* and *ExtInfo*. All data in the system is ontologically demarcated in OWL language (Web Ontology Language). Functionalities of the system are as follows:

1. *Data updates* – are divided in three groups: Checking Updates (algorithm to classify data to time sensitive or not), Known Updates (updates of time sensitive data) and Regular Updates (updates of no time sensitive data),
2. *New data* – Consistency checking (by using ontology reasoner Pellet – OWL Description Logics reasoner), Conflicts resolving (by calculating Certainty Factors) and Data saving,
3. *Incomplete data* (data incompleteness checking and requesting missing information).

All data manipulations are performed by using JENA (Java framework for building Semantic Web applications). Working status of the CMS can be tuned by changing the constant values in the configuration file of the CMS to achieve more efficient results of *Data updates*, *New data* processes or *Incomplete data* checking.

Keywords:

- Semantic Web
- Ontology
- Multi-agent System

Acknowledgements

I very thank my supervisor Dr. Marcin Paprzycki for his continuous support and guidance throughout the completion of this thesis. I would like to thank Maciej Gawinecki (from Systems Research Institute at the Polish Academy of Sciences), his ideas and technical insights in the first stage of this thesis were very valuable to me.

Paweł Olesiuk

Table of contents:

1. Introduction

At the beginning the World Wide Web (WWW) appeared as some interconnected computers intended to work together and share out the work (1989, Tim Berners-Lee). In its first stage the WWW was meant as the exchange of documents and data and some kind of working collaboration. Its purpose was to be a big working place where programs and databases could mutually share their knowledge and work. But with the explosion of personal computers and of the media programs, films, music, pictures, etc., the WWW is now almost only used by humans and not by machines. Machines cannot understand the real meaning of this data. This meaningless information is not useful at all for machines, which cannot operate with this data. The explosion changed the main assumptions of the WWW, when the idea of the one man is incredibly widespread used by humans from all world and developed by them.

The growth of the WWW has been impressive these last years. The amount of information in the WWW is huge and grows very fast. One of the biggest problems of handling this information is how to find what we are searching for – in other words, we face problem of the information overload. The project Semantic Web (1999, Tim Berners-Lee) and the software agents can be the answer for the arising problem. In the case of the Semantic Web the information is ontologically demarcated – we add semantic meaning to the data. According to P. Maes [Maes, 1994], intelligent software agents can be the solution for the information overload when the ontologically demarcated information will be managed by the intelligent agents – the machines will then be able to understand what is the real meaning of the data.

The present thesis focuses on travel-related information. A lot of travel-related data is available in the WWW, but in most cases this data is written only for human consumption. To check the hypothesis of P. Maes, the academic agent-based project Travel Support System (TSS) has come into being. TSS is the system for supporting travellers' needs. In the scope of travelling the design of the TSS includes such aspects as: standard transportation, choices of accommodation, restaurants, movie theatres, national parks, historical sites and other points of interest. The architecture of the TSS includes three subsystems: Content Collection Subsystem, Content Management Subsystem and Content Delivery Subsystem. The objective of this thesis is to develop the Content Management Subsystem, which would be mainly

responsible for managing the travel-related data in the storage of the TSS – data in the storage has to be current, reliable and complete. The data in the TSS is ontologically demarcated and all aspects of managing data are performed by software agents.

The thesis is organized as follows: Chapter 1 introduces the main objective of the thesis. Chapter 2 provides the Semantic Web technologies, including Multi-agent Systems. In Chapter 3, the current state of Travel Support System is provided. Chapter 4 includes analysis of the thesis's objective, created ontologies and the implemented system. Chapter 5 provides conclusions and possible technical restrictions in the future.

2. The Semantic Web

The phenomenon of the World Wide Web (WWW, Internet) is based on possibility of finding information practically on every topic in several seconds. Web browsers are irreplaceable in this respect. Web browsers search through millions of web pages and try to find what the user is interested in. But a web browser does not know, which of the found web pages include exactly what the user wanted to find. Web browsers work in a primitive way – trying to respond to the user's query by matching key words and ordering the result list of the web pages according to the number of key words on the web page or to popularity of the web page with respect to the number of links provided to it from other websites (and vice versa). However, very often the user is not going to find what they wanted. Furthermore, the results are whole web pages, and not detailed information. Therefore these pages can be understood only by a human, not by a computer (by a machine). Even in the case of a more advanced query, the current web browsers fail. Furthermore, web browsers cannot collect detailed information from several web pages. To justify this, let us consider two examples:

a) the word "cook" can be either a noun signifying a person who is making food or a verb signifying the method of making food, so if we query the web browser with the term "restaurant cook" (meaning a person), we will receive pages containing two meanings of this word or just one (other than we wanted to find) – in the first case we have information overload, it the second – lost information.
b) "*give me the cheapest transfer flight to Bangkok in the second half of July*" [Nowak, 2004] – nowadays web browsers cannot deliver information like this one.

To solve these problems users need something else than the presently available WWW architecture. To achieve this "something else" many people point to the Semantic Web project.

Semantics is the discipline of science dealing with relations between expressions (signs) and the meaning (things) they refer to. For example the statement "the windows clean restaurant" is correct syntactically, but incorrect semantically (doesn't have a sense).

The term "Semantic Web" was proposed in 1999 by Tim Berners-Lee[1], the inventor of the World Wide Web, in his book "Weaving the Web" [Berners-Lee, 1999]. The Semantic Web in this book is described as:

"The Web of data with meaning in the sense that a computer program can learn enough about what the data means to process it."

In its assumptions, the Semantic Web uses existing HTTP[2] protocol, that is, the same one on which the current WWW is based. But the difference lies in the fact that the sent information will be understood by machines. Understanding depends on the form of the sent data, in which the machines can convey the meaning between themselves. Furthermore, the Semantic Web allows both human users and machines to query the Internet as if it was a very large database.

Quite often, together with the Semantic Web terms like ontology and software agents are claimed to be closely connected [Hendler, 1999].

2.1 Architecture of the Semantic Web

When using the Semantic Web to send simple data A, we also put to it data B, which is information about data A. We can say that in the Semantic Web all sent information has information about itself (so-called metadata – data about data). This metadata includes expressions about relations between data and logic rules, which can be applied to this data. And this metadata will make it possible to understand data by machines. Machines will conclude real meaning of sent information. Thus we can treat mentioned metadata as semantic data.

Figure 2.1 presents architecture of the Semantic Web in the form of the layers of Web technologies and standards. The lowest layer, where the Unicode and URI (Uniform Resource Identifier) are located, makes sure that we use international characters sets (Unicode) and provides means for identifying the objects (URI), and the URI identifiers are constructed from Unicode. Next layer, XML (eXtensible Markup Language) with namespaces and XML

[1] Tim Berners-Lee is the Director of the World Wide Web Consortium (W3C), in 1989 he invented the World Wide Web, he wrote the first web client and server in 1990, his specifications of URIs, HTTP and HTML were refined as Web technology spread. Source: http://www.w3.org/People/Berners-Lee/
[2] Hypertext Transfer Protocol

Schema, provides common syntax that we can integrate the Semantic Web definitions with the other XML based standards. RDF and RDF Schema, which are located on the next layer, allow to use the URI with statements about objects and to define vocabularies that can be referred to by the URI. This layer together with the ontology layer defines relations between objects. The other advantage of such construction is possibility to describe world in a way which can be understood by computers. Understanding of data by machines will be accomplished by using logic rules to common deduction and proofs to come to a conclusion (layers above the ontology layer). And the highest layer, the trust, will make possible to confirm the credibility for drawn conclusions.

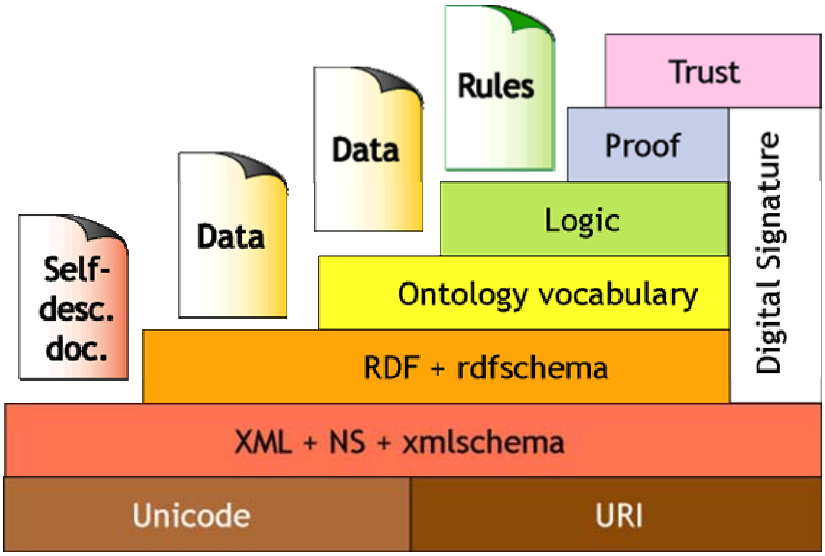

Figure 2.1 Architecture of the Semantic Web (source: "Semantic Web on XML" – slide Architecture, http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html)

The Digital Signature layer, which joins the four layers, is for detecting alterations to documents and establish how credible is given information, but this layer is currently under theoretical construction only.

2.2 Ontology

The term "ontology" was borrowed from philosophy but quickly established as a handy word for a new approach to creating abstractions needed when using computers for

real-world problems. In philosophy, ontology is the study of being or existence[3], it is trying to answer questions: what exists? and if what exists can be split, then what are the components and what kind of relations are between these components?. These questions highlight the most basic problems when building ontology: finding a subject, a relationship, and an object to talk about.

In Computer Science and Artificial Intelligence (AI) ontology means the specific method for the knowledge formalization[4]. The most popular definition, from an AI perspective, is given in [Gruber, 1992] as follows:

"An ontology is an explicit specification of a conceptualization,' where 'a conceptualization is an abstract, simplified view of the world that we wish to represent for some purpose."

When designing an ontology we have to use method like classification (Restaurant – the class of restaurants) and then order identified classes into a hierarchy (the classes: Chinese Restaurant, Polish Restaurant – are under Restaurant class in the hierarchical structure). Restaurant class is a concept, and the instance of the some class (Restaurant "El Popo") is also a concept. In the ontology we describe world by using concepts:

Restaurant "El Popo" offering nachos.

where *offering* and *nachos* are concepts, too.

2.2.1 Resource Description Framework

The World Wide Web is based on three primary components: HTTP, URLs (Universal Resource Locators) and HTML (Hypertext Markup Language). With the fast growing of WWW and use of HTML became widespread, Web developers came to point of limitations in HTML language. They found that the HTML language is not extensible and not useful with Web applications such as Web Services when exchanging data between services. To manage with restrictions of HTML, the next solution was XML. It provides a uniform framework for exchanging data between applications and a surface syntax for structured documents. But

[3] Source: http://en.wikipedia.org/wiki/Ontology
[4] Source: http://pl.wikipedia.org/wiki/Ontologia

XML imposes no semantic constraints on the meaning of these documents. To deal with the semantics of data, the project Semantic Web includes the Resource Description Framework.

Resource Description Framework[5] (RDF) was developed by the World Wide Web Consortium (W3C). RDF is a model of statements made about resources and associated URIs and this model provide a simple semantic and can be represented in XML syntax. Its statements have a uniform structure of three parts (known as triple): subject, predicate and object. To represent the statement from previous chapter by means of triple, it will be:

Triple (Restaurant "El Popo", offering, nachos)

where *Restaurant "El Popo"* is a subject, *offering* is a predicate, *nachos* is a object. The subject describes resource with associated URI, which can be any concept. The predicate (property), which again has own unique URI, is a characteristic of a subject or a relation between resources. The object (value of the property) can be a resource referred to by a predicate or a literal (text or number value).

We can visualize RDF statements with:

- RDF/XML[6] syntax or with notation N3[7] – flexible communication between application systems,
- RDF graph – convenient for communication between people; it is a directed graph, nodes represent subjects or objects, edges represent properties.

The code below shows our Restaurant example in the RDF/XML syntax.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:res="http://world-ontology.com/Restaurant#">
<res:Restaurant rdf:ID="http://polishrestaurants.com#ElPopo">
   <res:offering rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
   >nachos</res:offering>
</res:Restaurant>
</rdf:RDF>
```

[5] http://www.w3.org/RDF
[6] http://www.w3.org/TR/rdf-syntax-grammar/
[7] http://www.w3.org/DesignIssues/Notation3.html

In the RDF/XML code above, the URI of the restaurant *El Popo* (subject) is "http://polishrestaurants.com#ElPopo". The predicate *offering* has a URI "http://world-ontology.com/Restaurant#offering" (*res* is a namespace[8]). The object *nachos* is a literal which has a type *string*. Now let *nachos*, as a resource, are a popular snack food, originating in North America[9]. After changing to it, the RDF/XML code will look like:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:res="http://world-ontology.com/Restaurant#">
<res:Restaurant rdf:ID="http://polishrestaurants.com#ElPopo">
   <res:offering rdf:resource="http://world-ontology.com/Restaurant#nachos"/>
</res:Restaurant>
</rdf:RDF>
```

So now, *nachos* is a resource with the URI: "http://world-ontology.com/Restaurant#nachos" and *nachos* will become common understanding word between people and application systems.

Figure 2.2 shows possible shapes for visualization different components of the triples with the RDF graph.

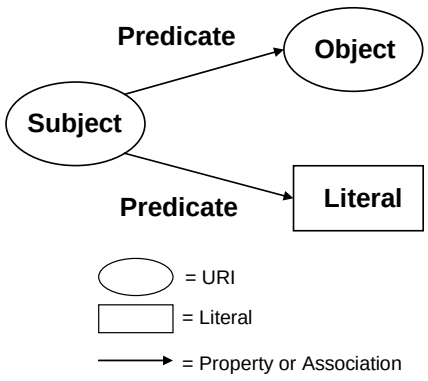

Figure 2.2 Shapes of components in the RDF triple (source: [Daconta, 2003]).

[8] In RDF is used namespace mechanism of XML. But in XML namespaces are used to remove ambiguities, in RDF namespaces are expected to be RDF documents defining resources, which are used to import RDF documents and we can retrieve additional information about resources (vocabularies – RDF Schema).
[9] Source: http://en.wikipedia.org/wiki/Nachos

The previous restaurant example, with extensions of predicates like *title, price, city*, *streetAddress*, is drawn in figure 2.3.

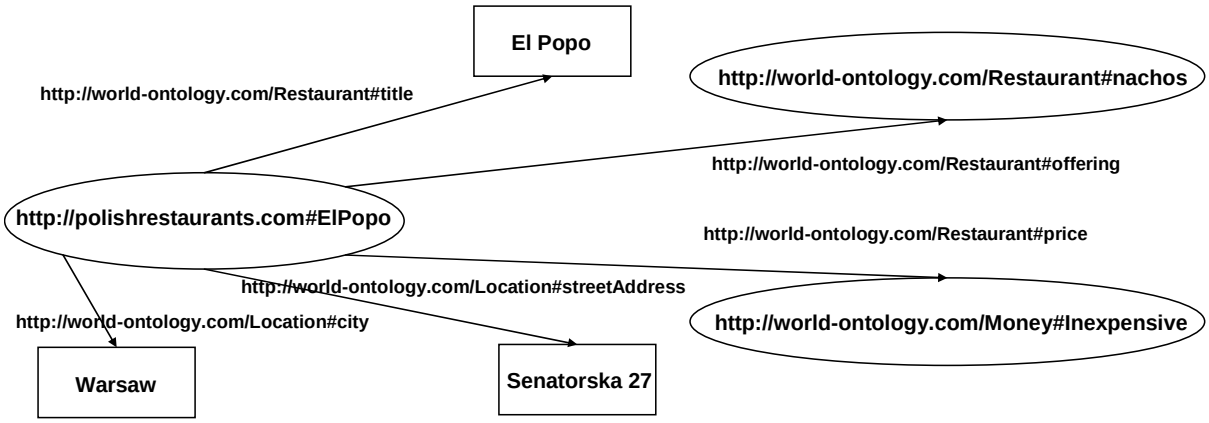

Figure 2.3 RDF graph – Restaurant "El Popo" example.

As we can see in figure 2.3, with others predicates we have new namespaces: "http://world-ontology.com/Location#" and "http://world-ontology.com/Money#". So now we have elements like *Restaurant*, *Location* and *Money*. And these elements are classes – sets of resources. We describe classes, their relationships with subclasses and properties associated with classes by using the RDF Schema (a vocabulary of RDF resources).

The RDF Schema (RDFS) is toward RDF, like XML Schema to XML. RDFS is describing the semantics for generalization-hierarchies of classes and properties – information of the information, or meaning of the information. RDFS is similar to Object-Oriented programming (OOP), in which we have a class, an instance of the class, fields and inheritance. When we want to describe a specific domain of the world, we specify the objects we want to talk about. We can talk about either individual objects (resources, instances of the class) or classes that define types of objects which have a common characteristic (fields, in RDFS – properties). The main difference between RDFS and OOP is that RDFS properties are defined globally and it is possible to define new properties of the class without changing that class.

The basic elements of RDF and it extension RDFS are expressed as:

- rdf:type – to declare that something is a type of something else,
- rdfs:Class – Class is a type of Class,
- rdfs:Resource – Resource is a type of Class,
- rdf:Property – Property is a type of Class,

- rdfs:subClassOf – the Class B is a subclass of the Class A,
- rdfs:subPropertyOf – the Property D is a subproperty of the Property C,
- rdfs:domain – restricts the set of classes that may have a given property (property domain),
- rdfs:range – restricts the set of classes (object type properties) or values (data type properties) for a given property (property range).

Reification

To conclude this chapter we have to introduce the term *reification*. The RDF allows us to make statements about statements using a reification mechanism, in the way that a statement is treated as a resource and hence the ability to make assertions about that statement. Reification is useful to describe belief or trust about statements. It can be also used to add some additional information to a given statement (who created the statement, when, etc.). For example if we have the statement *Triple (Restaurant "El Popo", offering, nachos)* and we will attribute to it a value X (it can be an URI or a bNode[10]) then we can build new statements about it: *Triple(X, confirm, ElPopoCook)* – that the statement X is confirmed by the cook of restaurant "El Popo" – *nachos* are really offered in this restaurant.

2.2.2 Web Ontology Language

Relation between RDF and RDFS can be compared to current WWW and Semantic Web, or XML and RDF. Unlike the latter, the first technologies mentioned have limitations in describing more useful ontologies for machines to perform automated reasoning. For example – they cannot describe simple constraints such as cardinality constraints. More sophisticated language, which allows put more details on ontologies, is the Web Ontology Language.

The Web Ontology Language (OWL) was defined in 2004 by World Wide Web Consortium and is based on the DAML+OIL[11] web ontology language, which OWL superseded. OWL is built on top of RDF. OWL extends the RDF with more vocabulary for describing properties and classes. The W3C has defined OWL to include three different

[10] bNode stands for "blank node", which refers to the fact that the corresponding nodes in the RDF graph are "blank" – have no label; source: http://www.w3.org/2005/rules/wg/wiki/bNode_Semantics

[11] DAML+OIL was developed from 2000 until 2006 by a group called the "US/UK ad hoc Joint Working Group on Agent Markup Languages" which was jointly funded by the US Defense Advanced Research Projects Agency (DARPA).

sublanguages in order to offer different balances of expressive power and efficient reasoning [Ontology Web Language Features]:

- OWL Lite – supports a classification hierarchy and simple constraints like definition of concepts through applying to them relations with cardinality values of 0 or 1,
- OWL DL – a superset of OWL Lite, supports the maximum expressiveness and enables to define complex concepts through applying to them various kinds of cardinality constraints on relations, but we can't define arbitrary relations between concepts, only between their instances – effect of this restriction is computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in a finite time); OWL DL is so named due to its correspondence with *Description Logics* (described in the next section), hence the suffix DL,
- OWL Full – a superset of OWL DL, supports the maximum expressiveness and the syntactic freedom of RDF but without above constraint there are no computational guarantees (no completeness or decidability).

OWL DL language extends RDFS by adding description logics expressiveness to it and allows to create relations between complex restrictions to class and property definitions. OWL DL extends RDFS in the following ways:

- additional restrictions on properties like: *allValuesFrom*, *someValuesFrom, hasValue* (which values can be used) or cardinality constraints: *cardinality*, *minCardinality*, *maxCardinality* (how many values can be used),
- definition of classes by enumerations of their instances: *oneOf* (the class DaysOfTheWeek has only 7 instances, no more, no less, which are days of the week),
- definition of classes by terms of other classes and properties (class expressions using *unionOf*, *complementOf*, *intersectionOf*),
- ontology and instance mapping (*equivalentClass, equivalentProperty, sameAs, differentFrom, AllDifferent*) permitting translation between ontologies,
- additional hints to reasoner (*disjointWith, inverseOf, TransitiveProperty, SymmetricProperty, FunctionalProperty, InverseFunctionalProperty*).

Some explanations with examples of these terms are provided in the next section. OWL DL is increasingly applied in practice, e.g. in systems like *KAON*[12], *Protégé*, *Jena* (two last are described in section 2.2.5 *Ontology tools*).

2.2.3 OWL Description Logics

Description Logic (DL) is knowledge-representation language adapted for expressing knowledge about concepts and concept hierarchies, and it is very well suited for providing structure to information. Description Logic is a decidable subset of First-order Logic[13] (FOL) and therefore is amenable to automated reasoning. It is possible to automatically compute the classification hierarchy[14] and check for inconsistencies in an ontology that conforms to OWL DL.

Some terms existing in both languages differ, for example: a *concept* in DL is referred to as a *class* in OWL, a *role* in DL is a *property* in OWL. The set of vocabularies, like concepts definitions with their limitations and relations between concepts, is called a T-Box (Terminological Knowledge). At the same time an A-Box (Assertion Knowledge) is a set of facts, such as definitions of particular instances of the concepts (individuals) and relations between those, associated with a terminological vocabulary. T-Box and A-Box are used to describe two different types of statements in ontologies. Together T-Box and A-Box statements make up a knowledge base (figure 2.4).

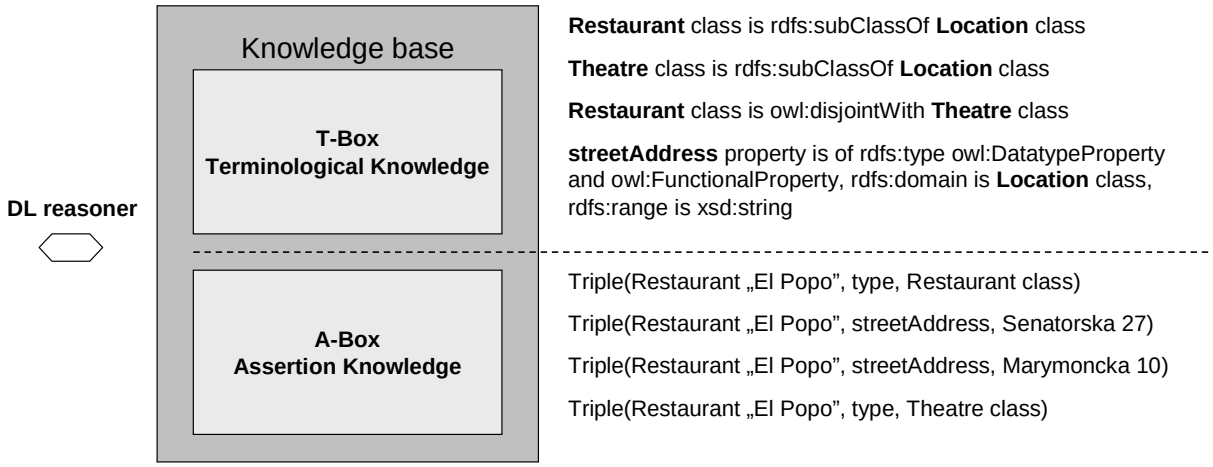

Figure 2.4 Knowledge base: T-Box and A-Box (source: [Lutz] with self examples)

[12] KAON (KArlsruhe Ontology) is an open-source ontology management infrastructure targeted for business applications, http://kaon.semanticweb.org/
[13] Logics are decidable if computations based on the logic will terminate in a finite time.
[14] Also known as subsumption reasoning.

The reasoner (inference engine) is a piece of software able to infer logical consequences (with respect to T-Box) from a set of asserted facts (A-Box). T-Box defines inference rules. Inference rules in ontologies may express rules for manipulating information in A-Box.

In figure 2.4 T-Box includes definitions: *Restaurant* is a subclass of *Location*, *Theatre* is also subclass of *Location*. *Restaurant* and *Theatre* are disjoint classes, so that an individual (or object) cannot be an instance of more than one of these two classes. Property *streetAddress* is *Datatype* and *Functional*, domain of it is *Location* class and range is string type. *Functional* property means that there can be at most one object that is related to the other object via this property. So the reasoner, e.g. can inference that *streetAddress* can also be used to instances of *Restaurant* class, because it is subclass of *Location* class.

How does reasoner work? Let us try to infer logical consequence from examples in an A-Box (in figure 2.4). The first triple is correct, we can say it is consistent. The reasoner will remember that object *Restaurant "El Popo*" belongs to the *Restaurant* class. The second triple is also consistent, *streetAddress* property inherited from *Location* class has string value and it is first *streetAddress* value for the *Restaurant "El Popo*". Now, the reasoner will infer that the third triple is inconsistent, because the *Restaurant "El Popo*" already has *streetAddress* value and the same physical restaurant can't be in two different places (*Functional* property). The fourth triple is inconsistent either, because the *Restaurant "El Popo*" can't be in two disjoint classes, it can't be a both restaurant and theatre in the same time.

OWL DL corresponds to the *SHOIN(D)* description logic. We can explain *SHOIN(D)* as an abbreviations of letters:

S – complex concept negation, concept intersection, universal restrictions with *Transitive* properties,

H – role hierarchy (*rdfs:subPropertyOf*),

O – enumerated classes of object value restrictions (*owl:oneOf*, *owl:hasValue*),

I – *Inverse* properties,

N - cardinality restrictions (*owl:cardinality*, *owl:maxCardinality*),

(D) - *Datatype* properties, data values or data types.

From the *Transitive* property reasoner can infer, that if object A relates to object B, and B relates to object C, then A relates to C.

2.2.4 SPARQL

The Semantic Web will allow to treat the web documents as if they were in a single big database (web documents demarcated using RDF and/or OWL). This big logical database will allow querying and manipulating data stored there. According to Tim-Berners Lee, using the Semantic Web without SPARQL (known as A-Box queries in DL terminology) is like using a simple relational database without SQL[15]. SPARQL's (Protocol and RDF Query Language) specification[16] was finally published by W3C on 15 January 2008. So it is quite young final specification, but when it had the working draft status, many projects were built with relation to the SPARQL; e.g. DBpedia[17], DBLP Bibliography[18], which allow for sophisticated querying using SPARQL Language.

To present the syntax of the SPARQL, listing 2.1 provides a simple query, which will give results about all restaurants (*names* and *streets addresses*) in the *city Warsaw* which are *offering nachos* and are *inexpensive*.

```
PREFIX res: <http://world-ontology.com/Restaurant#>
PREFIX loc: <http://world-ontology.com/Location#>
PREFIX money: <http://world-ontology.com/Money#>

SELECT ?restaurantTitle ?restaurantStreetAddress
WHERE {
        ?x rdf:type res:Restaurant
        ?x res:title ?restaurantTitle .
        ?x loc:streetAddress ?restaurantStreetAddress .
        ?x loc:city "Warsaw" .
        ?x res:offering res:nachos .
        ?x money:price money:Inexpensive .
}
```

Listing 2.1 Simple SPARQL query

[15] Structured Query Language
[16] http://www.w3.org/TR/rdf-sparql-query/
[17] http://wiki.dbpedia.org/About - DBpedia allows you to ask sophisticated queries against Wikipedia and to link other datasets on the Web to Wikipedia data.
[18] http://www4.wiwiss.fu-berlin.de/dblp/ - provides information about scientific publications.

Variables are indicated by the "?" prefix. Bindings for the ?restaurantTitle and the ?restaurantStreetAddress will be returned. The variable ?x corresponds to subjects (instances of restaurants, e.g. Restaurant "El Popo") in triples from some RDF database/catalogue that match given conditions (city "Warsaw", etc.). All titles and street addresses of restaurants from queried repository with respect to conditions will be returned.

2.2.5 Ontology tools

Protégé

The Protégé[19] is an ontology editor and knowledge-base framework, which was developed at the Stanford University School of Medicine in USA. With Protégé we can easily create classes with hierarchy of them, object properties (with almost all of OWL expressions like *Transitive*, *Symmetric*, *Irreflexive* or *cardinality*) and data properties, individuals (with *sameAs, differentFrom*), and then visualize our ontology with OWLViz or OntoViz plugins. It also has a DL Query plugin for quickly testing definitions of classes to see that they subsume the appropriate subclasses. Note also that Protégé supports ***SHOIN(D).***

Jena

Jena[20] is a Java framework, which allows programmers to create the Semantic Web applications. Jena is open source and grown out of work of the *HP Labs Semantic Web Programme*[21]. Jena provides very useful and fast interface for reading, writing and manipulating RDF and OWL documents (files). It also supports querying these documents by SPARQL query engine and to allocate RDF documents in persistent storage like relational databases: PostgreSQL, HSQLDB, MySQL, Oracle, Microsoft SQL Server. Furthermore, it includes a rule-based inference engine and a reification mechanism.

Pellet

Pellet[22] is an open source Java based OWL DL reasoner. It can be used in conjunction with Jena and is recommended by Jena. Pellet provides functionalities to see the species validation, check consistency of the ontologies, check entailments and answer a subset of

[19] http://protege.stanford.edu/
[20] http://jena.sourceforge.net/
[21] http://www.hpl.hp.com/semweb/
[22] http://pellet.owldl.com/

SPARQL queries. Pellet is based on the tableaux algorithms developed for expressive Description Logics. It supports reasoning with the full expressivity of OWL DL (*SHOIN(D)*) including reasoning about owl:oneOf and owl:hasValue. In this thesis Pellet is used for consistency checking of the ontologies.

Features of Pellet: Standard Reasoning Services, Datatype Reasoning, Conjunctive Query Answering, Rules Support, Ontology Analysis and Repair, Ontology Debugging, Incremental Reasoning.

2.3 Multi-agent System

The term *agent* is the next very important element used in the Semantic Web. According to it, an agent is a piece of software which will collect information from several web ontology sources, manage it and distribute for humans or other agents.

"One of the biggest problems we nowadays face in the information society is information overload. The Semantic Web aims to overcome this problem by adding meaning to the Web, which can be exploited by software agents to whom people can delegate tasks" [Esperonto Project].

Ontology specifies a conceptualization, it represents an abstract and simplified view (vocabulary, relationships and logical rules) of the piece of reality it wants to represent. Committing to ontology, agents will know which vocabulary they are referring to. With RDF we can express statements in a formal way that software agents can read and act on.

What really is a software agent? There is no one common definition of an agent [Paprzycki, 2003]. But we can define agent as a program that is situated in some environment, capable of communication, interacting with other agents or humans, monitoring its environment, taking autonomous decision and initiatives to achieve goals.

Wooldridge [Wooldridge, 1997] defines an agent as a system with the following characteristics:

- autonomy – agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state,
- reactivity – they perceive their environment and respond,

- pro-activeness – they perform a given task without stimulus from a human,
- social ability – agents interact with other agents (and possibly humans) via some kind of agent-communication language.

The complex of the agent, additional possibilities like ability to adapt or learn (intelligent agent), mobility, is dependent of the objective, for which it was designed.

When several agents aim at meet its design objectives and interact with others to accomplish the global objective, then they form a Multi-agent System (MAS). As example of developing a MAS is building/implementing a robot that is cleaning the house[23]. The easier solution is to build one small specialized robot/agent to the vacuum, separate to take out the trash, another separate to clean the windows, than to build one big robot/agent that will perform all mentioned tasks. Every robot/agent is performing only dedicated to him tasks, all robots work to achieve the global task/objective – to clean the house.

Above example also relates to the one of methodology when solving complex tasks – *decomposition* ([Jenings, 1999] with reference to the MAS from [Booch, 1994]). *Decomposition* is a division of a large problem to the smaller parts/problems, which can be implemented in the independent way (autonomous operations) from others.

All aspects of the term agent apply to the fairly new programming paradigm Agent-Oriented Programming (AOP). Agent-Oriented approach can be viewed as next step of Object-Oriented approach [Chavarkar] that supports a societal view of computation.

2.3.1 Agent communication

The Semantic Web requires that software agents communicate with each other, it is also as characteristic of an agent – social ability. The *Agent Communication Language*[24] (ACL) was proposed by the *Foundation for Intelligent Physical Agents*[25] (FIPA) as a standard language for agent communications. ACL relies on speech act theory describing the way that one agent sends ACL messages to another. The sequence of sent messages constitutes a conversation.

[23] Example from source: http://4programmers.net/Z_pogranicza/FAQ/Metodyki_programowania #id-Programowanie-agentowe
[24] http://www.fipa.org/repository/aclspecs.html
[25] http://www.fipa.org/specifications/index.html

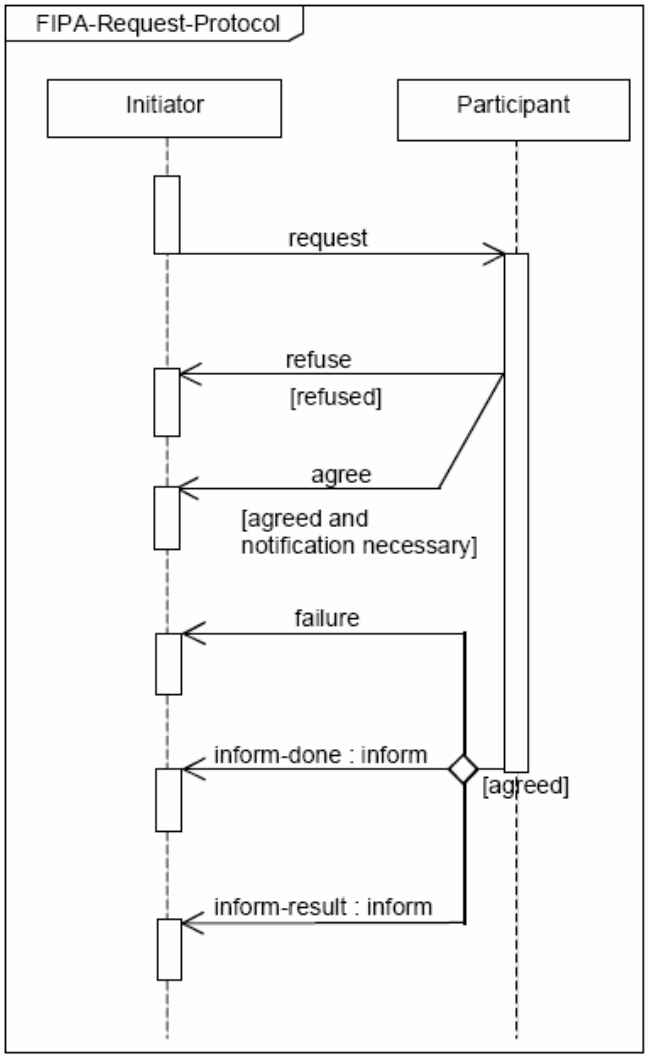

Figure 2.5 FIPA Request Interaction Protocol (source: [FIPA Request Interaction Protocol])

FIPA specified *interaction protocols* that define types of possible exchanged messages like *inform*, *request* or *propose* that form a pattern of interactions that can be applied to a specific situation. For instance, the FIPA Request Interaction Protocol[26] (figure 2.5) allows one agent – the *Initiator*, to *request* another – the *Participant*, to perform an action where the *Participant* processes the *request* and makes a decision whether to *refuse*, *agree*, raise *failure* or *inform*.

Even agent's speaking the same language (ACL) requires that agents will understand each other by using a common ontology that is a part of the agent's knowledge base. Ontology, which can be built in e.g. FIPA Semantic Language (FIPA-SL), describes the domain of things that agent can deal with and how they are related to each other.

[26] Another interaction protocols: FIPA Cancel Meta-Protocol, FIPA Contract Net Interaction Protocol

FIPA specified also elements like:

- White Pages – services for getting the identification number of agent by giving the name of it, it is useful to establish a communication between agents,
- Yellow Pages – agents register they own services that can perform for request others,
- Message Transport Service – service that provides transport of ACL messages,
- Agent Management System (AMS) – mechanism for creating, removing and managing inspection of agents.

2.3.2 Agent platform - JADE

In order to Agent-Oriented Programming becomes widespread, the community of computer programmers need useful tools to develop application systems with its. *Java Agent DEvelopment framework*[27] (JADE) is one of these tools. JADE is an open source platform fully implemented in Java[28] and is consistent with FIPA specifications; e.g. JADE supports ACL messages and ontology.

Creating an agent with JADE relies on defining a simple Java class that extends the core *Agent* class of JADE and implementing its behaviours. In this class we have to implement *setup()* method (initialization of an agent and behaviours) and *takeDown()* method. Behaviours are instances of the class *Behaviour* and their subclasses with *action()* and *done()* methods. In the *action()* method we define tasks that the behaviour is performing, method *done()* is informing if behaviour finished all delegated tasks and then the particular behaviour is deleted from the pool of active behaviours of the agent. In JADE the sequence of all initiated behaviours of the agent is managed by JADE mechanism presented in figure 2.6.

[27] http://jade.tilab.com/
[28] http://java.sun.com/

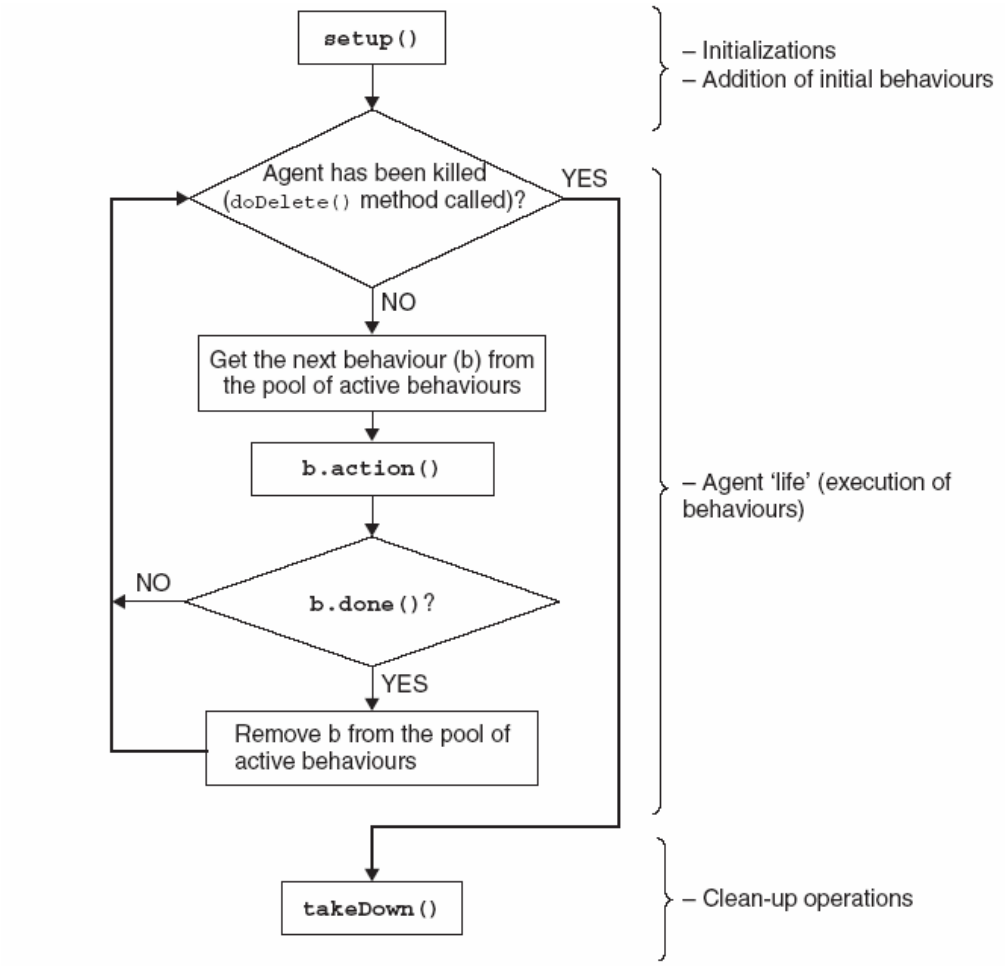

Figure 2.6 Agent thread path of execution (source: [Bellifemine, 2007])

JADE features:

1. Distributed agent platform – agents can work on different platforms with different OS[29] by connecting them using Java RMI[30]. Agents are implemented as Java threads.
2. Graphical user interface to manage agents (with *Sniffer Agent* – documenting conversations between agents, *Introspector Agent* – debugging the behaviour of an agent, *Dummy Agent* – simulation of ACL messages).
3. Efficient transport of ACL messages (experimentally proved in [Chmiel, 2004]).
4. Very useful implementations of behaviour classes like *CyclicBehaviour*, *TickerBehaviour*, *ThreadedBehaviourFactory,* or for complex task like *SequentialBehaviour*, *FSMBehaviour*[31].

[29] OS - Operating System
[30] Java Remote Method Invocation
[31] Finite State Machine Behaviour

3. Travel Support System

It is widely believed that the Semantic Web has an enormous potential. However, sometimes theory does not work out in practice. To check and verify the power of the Semantic Web and Multi-agent Systems the academic agent-based project Travel Support System[32] (TSS) has come into being.

3.1 TSS objective

The objective of the Travel Support System is to support needs of travellers. In the scope of travelling the design of the TSS includes aspects like the standard transportation, choices of accommodation, restaurants, movie theatres, national parks, historical sites and other points of interest [Angryk, 2002]. The aspect of the content personalization is also very important element in the TSS, to provide information that is in some range suit to user preferences.

Information used in TSS is stored in the central repository and it is the ontologically demarcated information. All semantic information is gathered from the Internet sources. Information is managed by agents.

3.2 TSS architecture

The project has started in 2001 [Ali, 2001. Galant, 2002. Angryk, 2002] and has evolved as time passed. During development of the project in [Gordon and Paprzycki, 2005] architecture of the TSS was presented and it is shown on figure 3.1. It is divided into three functional parts:

- Content Collection – collection of information from Verified Content Providers (VCP) or other Internet sources,
- Content Management – checking for consistency, completeness, deal with conflicts and timeliness of information,
- Content Delivery – delivery for users personalized information that they are searching for.

[32] http://agentlab.swps.edu.pl/agents_TSS.html

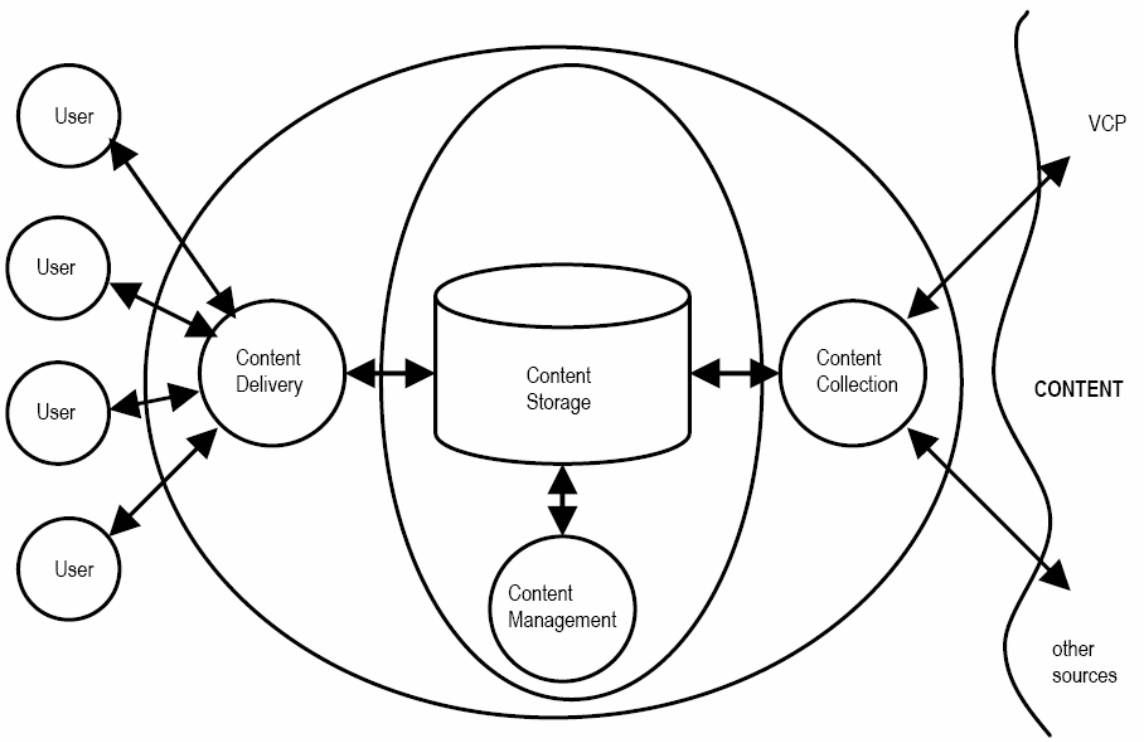

Figure 3.1 Infrastructure for the TSS (source: [Gordon and Paprzycki, 2005]).

3.2.1 Content Collection Subsystem

The Content Collection Subsystem (CCS) is responsible for delivering information from the Internet data sources in the semantic form (RDF, OWL). In CCS are distinguished two kinds of data sources:

- Verified Content Providers – information from them is reliable, available all the time and with the high probability they provide true information; to this group belongs very popular web portals or Internet sources that make available data in RDF form,
- other Internet sources – information from those are incomplete, sometimes contradictory from VCP sources, very often the form of data in which they store is changed.

VCP data sources can turn into other Internet sources, and vice-versa. It depends on quality of data that specific data sources provide. But nowadays there aren't such declared VCP sources on the Internet.

The TSS was originally designed to collect indices to data (by *Indexing Agents*), rather than data itself. But this approach had been changed. Information is saved in the Content Storage (JENA with persistent storage utility – OWL/RDF documents in the relational database). The current version of the TSS (TSS 1.0[33]) includes information about 9 thousand restaurants (which are in OWL language, converted from RDF) from *ChefMoz dining guide*[34] project. "The *ChefMoz* Project's goal is to produce the most comprehensive guide to restaurants, by relying on a vast army of volunteer editors"[35]. *ChefMoz* data is available in RDF/XML form, so there exists a big repository of information about restaurants, which can be read by machines. But this information was not semantically and syntactically correct, additionally it does not represent a specific ontology. Syntactic correctness and ontology was achieved by [Gawinecki, 2005a], [Gawinecki, 2005b].

CCS includes a set of *Wrapper Agents* that try to gather data from Internet sources [Pisarek, 2005]. In the current state of the Internet, where data in RDF seems to be like the needle in a haystack (except the *ChefMoz*), *Wrapper Agents* must extract HTML content from existing web sites into RDF triples "manually" [Gordon and Paprzycki, 2005]. This approach force to create distinct *Wrapper Agents* for particular web sites. This inconvenience will disappear when the Semantic Web will become widespread and RDF data will be widely available.

3.2.2 Content Management Subsystem

This thesis is about this subsystem, Content Management Subsystem (CMS). It is mainly responsible for:

- checking for consistency and conflicts of new information provided by the CCS according to the existing data and ontology rules,

- updating data stored in the central repository for information which can be time sensitive (e.g. cinema programs change on Fridays) or normal updates take in regular periods of time and are to assure data correctness,

- checking for incomplete information (e.g. telephone number) and try to get them from the CCS.

[33] http://agentlab.swps.edu.pl/agent_papers/tss-1.0-all.zip
[34] ChefMoz. 2005.*ChefMoz dining guide*: http://chefmoz.org/.
[35] http://chefmoz.org/about.html

CMS subsystem is the least defined subsystem in the current TSS design. In this thesis is made extensions which are described in section *4 Ontology Management System*.

3.2.3 Content Delivery Subsystem

This subsystem (CDS) delivers personalized information to the user [Gawinecki, 2005] (based to the query [Kaczmarek, 2005]). It is the most extended subsystem in the current version of the TSS. User can connect to the TSS by a web browser, register in the TSS (figure 3.2) and then (s)he can search restaurants by giving details of what (s)he wanted to find (figure 3.3). During the registration, user has to provide personal information, like age, wealth, profession. Every query made to the TSS will attempt at addressing personal preferences of the user.

Figure 3.2 TSS 1.0 – registration form (authors: Maciej Gawinecki, Paweł Kaczmarek)

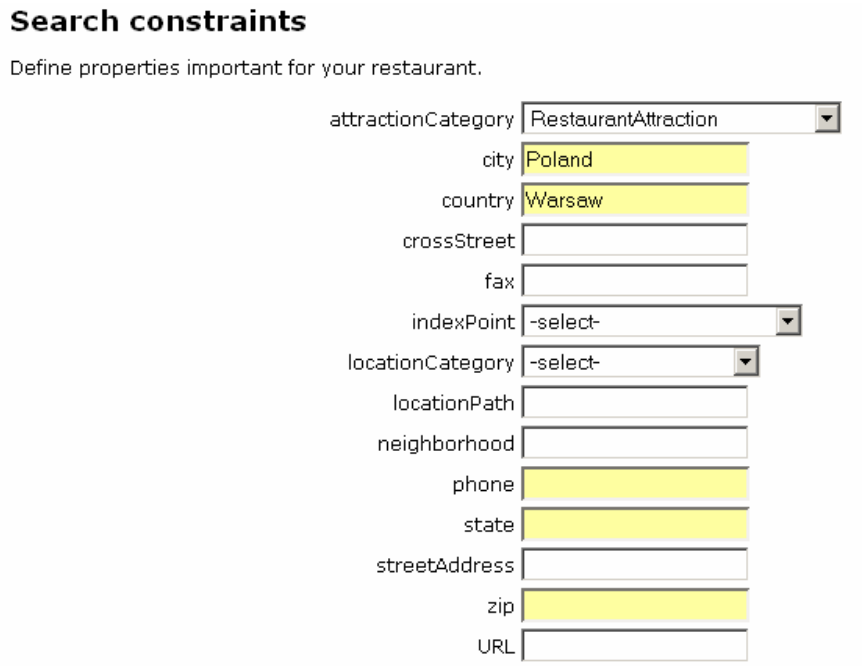

Figure 3.3 TSS 1.0 – part of search form (authors: Maciej Gawinecki, Paweł Kaczmarek)

The most important agents in the CDS are:

- *Proxy Agent* (*PrA*) – receive request from user and translate it into ACL message, then transfer ACL message to *Personal Agent* and get back results from it, send response to the user,
- *Personal Agent* (*PA*) – assures personalized results of user query with respect to the user profile,

Figure 3.4 TSS 1.0 – view in web browser of a found restaurant with its details (authors: Maciej Gawinecki, Paweł Kaczmarek)

- *Profile Managing Agent* (*PMA*) – is responsible for initializing and learning user profile on the basis of user feedback, it provides a user profile to the *PA* [Gawinecki, 2007],
- *View Transforming Agent* (*VTA*) – is responsible for conversion of semantic data (RDF) into HTML/WML/TXT documents, which can be viewed in convenient way by human on popular devices like web browser (figure 3.4) or mobile phones.

4. Ontology Management System

Objective of this thesis is to develop the Content Management Subsystem that is a part of the Travel Support System. Let us recall, what main roles it has to fulfill:

- data's consistency and conflicts checking,
- updates of stored data,
- stored data incompleteness checking and requesting missing information.

At present in the Travel Support System we deal with domain of restaurants, but my aim was to build universal application for any ontology system that will meet all above requirements. For the purpose of this thesis I have designed and implemented *Ontology Management System (OMS)*.

4.1 OMS ontologies

After the analysis of requirements of the CMS, I defined the ontology for the OMS. This ontology has two classes. First of them is a class *OntologySource* in which instances of ontology data sources are stored. Every ontology data source has a unique URI identifier. Listing 4.1 presents the defined ontology in OWL language.

```
<?xml version="1.0"?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns="http://www.ibspan.waw.pl/tss/OMS#"
  xml:base="http://www.ibspan.waw.pl/tss/OMS">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="OntologySource"/>
  <owl:Class rdf:ID="ExtInfo"/>
  <owl:ObjectProperty rdf:ID="hasPositiveOntSrc">
    <rdfs:domain rdf:resource="#ExtInfo"/>
    <rdfs:range rdf:resource="#OntologySource"/>
  </owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:ID="hasNegativeOntSrc">
  <rdfs:domain rdf:resource="#ExtInfo"/>
  <rdfs:range rdf:resource="#OntologySource"/>
</owl:ObjectProperty>
<owl:FunctionalProperty rdf:ID="hasMainOntSrc">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
  <rdfs:domain rdf:resource="#ExtInfo"/>
  <rdfs:range rdf:resource="#OntologySource"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="hasWrongTripples">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
  <rdfs:domain rdf:resource="#OntologySource"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="hasCorrectTripples">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
  <rdfs:domain rdf:resource="#OntologySource"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="hasURL">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
  <rdfs:domain rdf:resource="#OntologySource"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="hasUpdateDateTime">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#ExtInfo"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="hasIncompleteCheckDateTime">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
  <rdfs:domain rdf:resource="#ExtInfo"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="hasCF">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#ExtInfo"/>
```

```
        <owl:Class rdf:about="#OntologySource"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
 </owl:FunctionalProperty>
</rdf:RDF>
<!-- Created with Protege (with OWL Plugin 3.2.1, Build 365)  http://protege.stanford.edu -->
```

Listening 4.1 OMS ontology with two classes *OntologySource* and *ExtInfo* (OWL language)

OntologySource class includes following properties:

- *hasURL* (type xsd:string) – URL of the ontology source (it can be URL of the RDF documents or the web page from which *WrapperAgent* is parsing information and translating into RDF),
- *hasCorrectTripples* (type xsd:int) – number of consistent and non conflicting triples according to the stored data,
- *hasWrongTripples* (type xsd:int) – number of inconsistent and conflicting triples according to the stored data,
- *hasCF* (type xsd:float) – CF is Certainty Factor[36] that express how reliable ontology data source is and is computed using the formula:

$$hasCF = \frac{hasCorrectTripples - hasWrongTripples}{hasCorrectTripples + hasWrongTripples}$$

Certainty Factor has range of values from -1.0 to 1.0 and "main points" of this range can be translated into:

-1.0 ≡ definitely false/not reliable
-0.5 ≡ probably false/not reliable
0.0 ≡ unknown
0.5 ≡ probably true/reliable
1.0 ≡ definitely true/reliable

The greater the CF value is the more reliable the ontology source is. This value is useful to resolve conflicts between two ontology sources which provide e.g. different street

[36] David McAllister, Massachusetts Institute of Technology, http://www.rattlesnake.com/notions/certainty-factors.html

addresses for the same restaurant. The OMS will trust more ontology source with greater CF and from it will extract a street address for a particular restaurant and input into the data storage. It is also useful when defining if the ontology source is a Verified Content Provider or the other Internet source (see chapter 3). Computer program/machine can decide to which group given ontology source belongs – a human intervention is not required. For instance, it can be suggested that VCPs can be ontology sources with CF value greater or equal to 0.5, below this value are the other Internet sources. But, currently, in the OMS there is no distinction of ontology sources like these two kinds, only the CF value. Below is an example of the *OntologySource* class in RDF/XML syntax:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:oms="http://oms-ontology.com/OMS#">
<oms:OntologySource rdf:ID="http://oms.com#http1000chefmoz1110org">
   <oms:hasURL rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
   > http://chefmoz.org</oms:hasURL>
   <oms:hasCorrectTripples rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
   >260000</oms:hasCorrectTripples>
   <oms:hasWrongTripples rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
   >0</oms:hasWrongTripples>
   <oms:hasCF rdf:datatype="http://www.w3.org/2001/XMLSchema#double"
   >1.0</oms:hasCF>
</oms:OntologySource>
</rdf:RDF>
```

The second class in the OMS is the *ExtInfo* (*Extended Information*). With the *ExtInfo* class we add to triples/statements additional information by using the reification mechanism (which is built in JENA). Every statement is reified and marked by URI identifier. Instances of *ExtInfo* class have the same URI as the reified statement but have different name space (that corresponds to *ExtInfo* class). This class includes properties:

- *hasMainOntSrc* (object type property) – refers to the instance of the *OntologySource* class, states the main ontology source from which given statement was provided, it is a *Functional* property, so a given reified statement has only one value of this property,
- *hasPositiveOntSrc* (object type property) – refers to the instance of the *OntologySource* class, states the ontology sources that provided equal statements to

the given one; it is not a *Functional* property, so a given reified statement can have several of them, but different from the main ontology source,

- *hasNegativeOntSrc* (object type property) – refers to the instance of the *OntologySource* class, states the ontology sources that provided different statements to the given one, it is also not a *Functional* property,
- *hasCF* (type xsd:float) – Certainty Factor computed from the formula (the value 1 is added in the formula, because it one has the main ontology source):

$$hasCF = \frac{(count(hasPositiveOntSrc) + 1) - count(hasNegativeOntSrc)}{(count(hasPositiveOntSrc) + 1) + count(hasNegativeOntSrc)}$$

- *hasUpdateDateTime* (type xsd:string) – configuration of update date and time with addition of parameters in form A,B,C,D, where: A – type of update, B – value from last update (e.g. 1 is equal one day), C – value to next update (e.g. 7 means that next update will be in seven days), D – number of iteration; this property can include several update configurations, more clearly these parameters are described in section *Checking updates*,
- *hasIncompleteCheckDateTime* (type xsd:string) – date and time for checking incomplete objects (restaurant); it is initialized only to one main statement which represent particular object and includes one parameter – value to next update (1 is equal one day).

Here is an example of the *ExtInfo* class in the RDF/XML syntax (without the *hasPositiveOntSrc* and the *hasNegativeOntSrc* properties):

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:oms="http://oms-ontology.com/OMS#">
<oms:ExtInfo rdf:ID="http://oms.com#ReifStmt_70916423_1202559529921">
   <oms:hasMainOntSrc rdf:resource="http://oms.com#http1000chefmoz1110org"/>
   <oms:hasCF rdf:datatype="http://www.w3.org/2001/XMLSchema#double"
   >1.0</oms:hasCF>
   <oms:hasUpdateDateTime rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
   >2008-03-10 15:10:02;1,1,7,0&</oms:hasUpdateDateTime>
   <oms:hasIncompleteCheckDateTime rdf:datatype="http://www.w3.org/2001/
```

XMLSchema#string"
>2008-03-10 15:10:02;1</oms:hasIncompleteCheckDateTime>
</oms:OntologySource>
</rdf:RDF>

The explanations of using values of the properties from both examples of the classes are provided in section 4.3 *OMS utilities*. We can say that in OMS we deal with three ontology models: the *Restaurant* model, the *OntologySource* model and the *ExtInfo* model.

4.2 OMS agents

OMS includes several agents that co-operate which each other to meet requirements related to functionalities of the CMS. Several agents form the Multi-agent System that, according to the *decomposition* methodology, work to achieve the main objective of the CMS – data in the Content Storage kept up to date, reliable and complete. Agents are activated as a single instance or as multi instances. In the OMS we have implemented the following agents:

DBAgent

Only one agent, the *DBAgent* has connection to the Content Storage. It performs all operations of reading and saving data in the Content Storage. For reading the data it uses SPARQL language, for saving the methods implemented in JENA. Behaviours of the *DBAgent*, like *GetModelsForUpdate*, *GetIncompleteData*, *SaveNewData*, work in parallel by using predefined JADE behaviour *ThreadedBehaviourFactory* (behaviours as Java threads). There is no direct connection (communication) between the *DBAgent* and the *Ontology Providers*, all communication and data transfer is performed by others agents which perform tasks delegated to them.

SearchAgent

The *SearchAgent* is responsible of searching and providing ontology information from data sources (*Ontology providers*). *SearchAgents* are initialized in multiple instances by the *UpdateAgent* and the *IncompleteDataAgent*. Each initialized *SearchAgent* connects to a single *Ontology provider* and tries to extract from it the requested information.

NewDataAgent

New ontology data is received by the *NewDataAgent* from *Ontology providers*. New data is sent to the *DBAgent* which performs all aspects of saving new data to the Content Storage.

UpdateAgent

UpdateAgent works like a scheduled job in the database. By using predefined behaviour (a *CyclicBehaviour* of JADE) it checks constantly whether the data in the Content Storage should be updated. If it should, it takes data from the *DBAgent* and initializes *SearchAgents*, each for the specific *Ontology provider*, to get new values of needed data. New values are sent to the *DBAgent* which checks possible data changes and performs updates.

IncompleteDataAgent

IncompleteDataAgent works analogously to the *UpdateAgent*. The difference is that it checks date and time to search incomplete data in all *Ontology providers* included in the *OntologySource* model. It also initializes *SearchAgents* for specific *Ontology providers* if they already have not been initialized by the *UpdateAgent*.

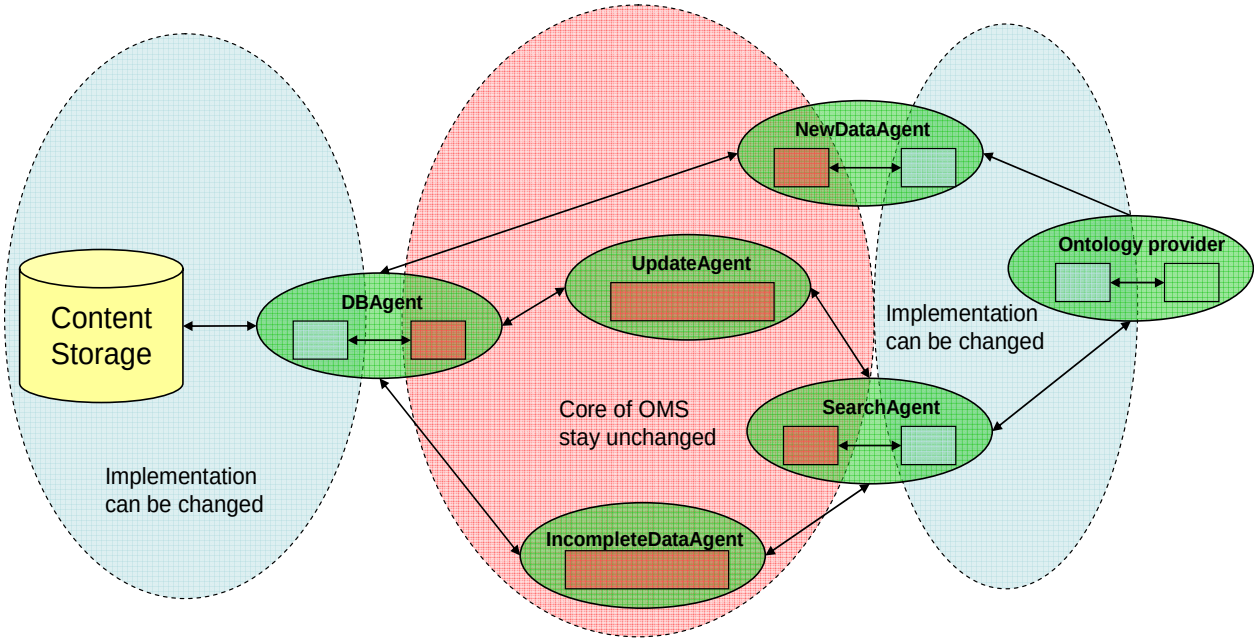

Figure 4.1 Interaction/communication between agents; green ellipses are agents, rectangles are piece of software of the agents.

Figure 4.1 presents interactions between agents. In the OMS all agents from this figure except of *Ontology provider* agents are implemented. This is because the letter agents belong to the Content Collection Subsystem. The figure also shows possible changes of the OMS implementation when the *Ontology provider* agents or the Content Storage change. As we can see, the core of the OMS in this situation does not require any changes, only parts of *NewDataAgent* and *SearchAgent* or *DBAgent* can be changed if needed. This illustrates the power of the Multi-agent System design [Jennings, 2001].

4.3 OMS utilities

Functionalities of the OMS are described in sections: *Data updates*, *New data* and *Incomplete data*.

4.3.1 Data updates

Data updates are divided into three groups: (a) Checking Updates, (b) Known Updates, and (c) Regular Updates. Checking updates are performed for new received data to classify it to known updates or regular updates. Known updates (scenario updates) are time sensitive, e.g. given statement changes in known times with a constant interval. Regular updates are performed to assure that stored data is up to date.

Checking Updates

Characteristic of data updating for this group is based on time variation updating. The method is used to check whether data is prone to frequent changes or not (because it is permanent in longer periods of time). We can presuppose that our property (datatype or object) varies often (e.g. names of different special parties at a restaurant which are organized periodically on every Thursday). However we cannot predict which properties could be preserved in this way and which need a frequent update. Moreover, our property could be preserved in a different way for a particular restaurant, for example the property *recommendedDishes* can have constant value for one restaurant, while for another restaurant it can change once a week or even every day. To define characteristics of object variability for different properties a method that can classify property frequency of changes, without human interaction, is proposed.

The method is initiated when *DBAgent* receives new information (a new restaurant with properties, new – missing earlier – properties for known restaurant) to check if given information is changeable in constant period of time, is sometimes changeable or not at all changeable (as street address of particular restaurant that can change extremely rarely or not at all). The method relies on initial checking every new statement once a week at the same time to check if there have been any changes. First the statement is reified and is added as an instance to the model *ExtInfo*. Overall, the property *hasUpdateDateTime* is set seven update configurations for consecutive days (schedule of updates) in the form:

date and time|typeUpdate;sinceLastUpdate;nextUpdate;iterationNo

where:

- *date and time* is the time point at which UpdateAgent will perform update checking (default value is date and time when statement was received plus one day),
- *typeUpdate* is a type of update, it can be *Checking_Update*, *Known_Update* or *Regular_Update* (default type is *Checking_Update*),
- *sinceLastUpdate* is a time interval from last update with unit of measure 1 = one day, 24 hours (0.5 is equal half of day, 12 hours, default value is 1),
- *nextUpdate* is a time interval when the next update should be perform (e.g. 7 = in 7 days, default value is 7),
- *iterationNo* is a number of iteration checking needed for changing the update group for given statement (e.g. after 3 iterations statement is moved to Known Update or Regular Updates groups, default value is 0).

Presented algorithms that deals with 'Time Adaptive' capability of data updates has been designed and implemented based on earlier work of presented in [Muthukumaraswamy Karthik, 2005], where it was determined how often data updates should be performed. According to this work, when process of data updating will return value TRUE – there were new values originating from the source – the frequency of visits value for the source is increased; if process of data updating will return FALSE – there were no changes of values in the source – the frequency of visits value for the source is decreased.

The method checks if statement is changeable once a week, once every two weeks or once a month according to the schedule of updates. Before presenting the algorithm of this method for group Checking Updates let us described used variables:

- *stmtEquals* – existing statement is equal to received statement after update process, no replacement,
- *stmtNotEquals* – existing statement is not equal to received statement, statement is replaced,
- *zeroIteration* – first iteration = 0,
- *finiteIteration* – last iteration = 3,
- *maxNextUpdate* – maximum value of the next update = 28, (we can say one month),
- *updateConf* – new update configuration.

Algorithm of the Checking Update method:

1. If *iterationNo=zeroIteration* and *stmtEquals* and *hasUpdateDateTime* property is empty and *nextUpdate>=maxNextUpdate* then new value of update configuration is:

   updateConf = (date and time)+ nextUpdate| Regular_Update; maxNextUpdate; maxNextUpdate; zeroIteration

   so given statement is not changeable and it is moved to the Regular Updates group.
2. If *iterationNo=zeroIteration* and *stmtEquals* and *hasUpdateDateTime* property is empty and *nextUpdate<maxNextUpdate* then new value of update configuration are new 7 update configurations with *sinceLastUpdate*2* and *nextUpdate*2*, so now the statement will be checked if it is changeable once in two weeks, furthermore once a month.
3. If *iterationNo=zeroIteration* and *stmtEquals* and *hasUpdateDateTime* property is not empty then update configuration is deleted from *hasUpdateDateTime* property.
4. If *iterationNo<finiteIteration* and *stmtNotEquals* then it is added new configuration:

   updateConf = (date and time)+ (nextUpdate- sinceLastUpdate/2) | Checking_Update; sinceLastUpdate/2; nextUpdate; iterationNo++
5. If *iterationNo>zeroIteration* and *iterationNo<finiteIteration* and *stmtEquals* then it is added new configuration:

   updateConf = (date and time)+ sinceLastUpdate/2 | Checking_Update; sinceLastUpdate/2; nextUpdate; iterationNo++

6. If *iterationNo=finiteIteration* and *stmtNotEquals* then it is added new configuration:

   updateConf = (date and time)+ nextUpdate | Known_Update;

   *sinceLastUpdate*2^ iterationNo; nextUpdate; zeroIteration*

   the statement is moved to Known Update group (statement is changing in known time point with constant interval)
7. If *iterationNo=finiteIteration* and *stmtEquals* then it is added new configuration:

   updateConf = (date and time)+ sinceLastUpdate | Known_Update;

   *sinceLastUpdate*2^ iterationNo; nextUpdate; zeroIteration*

   again the statement is moved to Known Update group.

Explanation of the course of the algorithm is provided with the following scenario: *DBAgent* received new statement S1 at the date time 2008-03-10 23:00:00 (figure 4.2). The statement S1 was reified and added (instance of it) to the *ExtInfo* model with the value of *hasUpdateDateTime* property equal to:

hasUpdateDateTime = 2008-03-11 23:00:00|1;1;7;0&2008-03-12 23:00:00|1;1;7;0& 2008-03-13 23:00:00|1;1;7;0&2008-03-14 23:00:00|1;1;7;0&2008-03-15 23:00:00|1;1;7;0& 2008-03-16 23:00:00|1;1;7;0&2008-03-17 23:00:00|1;1;7;0

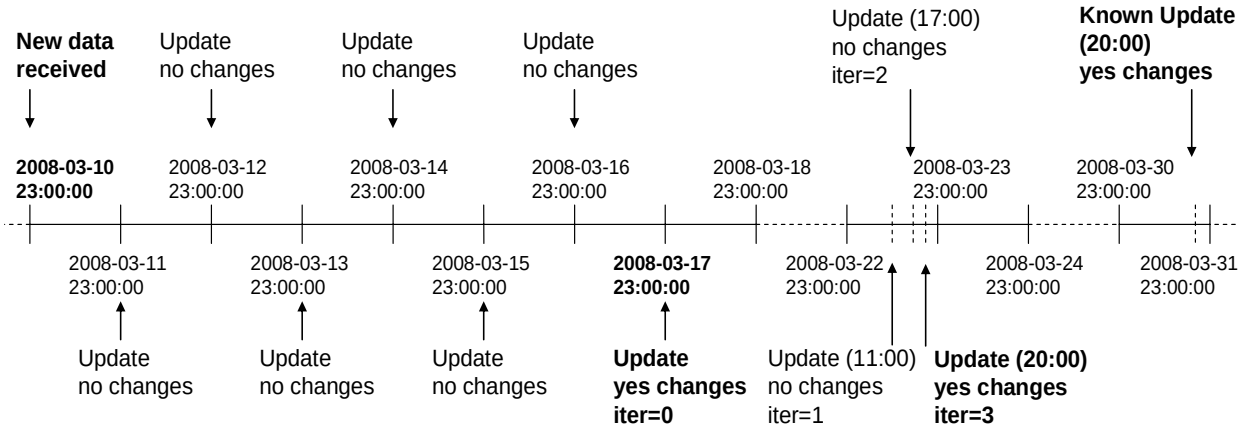

Figure 4.2 Checking Update example

Over the next six days *UpdateAgent* was checking if statement S1 has changed. It did not, thus during the sixth update configurations were deleted from the *hasUpdateDateTime* property (item 3 of the algorithm). On the 7th day, 2008-03-17 23:00:00, *UpdateAgent* received new value for S1 after updating. The old value was replaced in the storage data by a

new one. So we now that data changing takes place between 2008-03-16 23:00:00 and 2008-03-17 23:00:00, 24 hours of time. To come nearer to the time point of the data changing value 24 hours is dived by 2 and new update configurations became 2008-03-23 11:00:00|1;0.5;7;1 (the same day after 7 days subtract 12 hours, item 4 of algorithm). In the new configuration *UpdateAgent* checks S1 again but there were no changes. It is not a zero iteration, so we now that there was some data changing in the following 12 hours. Instead of checking data in 12 hours, this time is again divided by 2 and update configuration is 2008-03-23 17:00:00|1;0.25;7;2 (0.25 = 6 hours are added, item 5). After checking at a new time, the statements were equal again and a new update configuration became 2008-03-23 20:00:00|1;0.125;7;3 (6 hours are dived by 2 and 3 hours are added, item 5). At this timestamp statements were not equal, S1 was replaced, and it is finite iteration (*iterationNo*=3) so statement S1 moved to Known Update group with the update configuration 2008-03-30 20:00:00|2;1;7;0 (item 6). The result of the algorithm is that the given statement is changing on particular ontology data sources between 17:00 and 20:00 hour once a week. The statement is time sensitive and will be updated at this time once a week.

What will happen if statements will be equal? Again the statement will be checked on the primary hour, 2008-03-23 23:00:00|2;1;7;0 (item 7), as the Known Update. The section *Known Update* provides further sequence of events.

The advantages of this method are that we have very latest data in the storage with the error of 3 hours and when the data in the storage is more up to date, the dedicated system (Travel Support System) is more reliable.

Known Updates

Dates and times of updating statements in this group are known and performed in constant time intervals. Dates and times are the results of the method from the Checking Updates group. Given statement can be checked every two specific days (e.g. on Wednesday and Friday), every day of week, or some days of the month. If after updating there is a new value for the statement, the statement is replaced and a new update configuration is:

updateConf = (date and time)+ nextUpdate | Known_Update;
sinceLastUpdate; nextUpdate; zeroIteration

So the next update is going to be performed on the same day plus the value *nextUpdate* (it can be 7 days or 14 days) and at the same time. If the update returned negative result – statements are equal, then statement is again moved to the Checking Update group with the update configuration:

*updateConf = (date and time)+ sinceLastUpdate/*2 | *Checking_Update; sinceLastUpdate; nextUpdate; zeroIteration*

This is to check that maybe in the ontology data source statements are changing at a different time, e.g. if earlier known update time was hour 20:00, in ontology source given statement can change after this hour (e.g. at 21:00). But, if there were no changes after the method from the Checking Update group, the statement is no longer time sensitive, and it is moved to the group Regular Updates.

Regular Updates

Regular updates are performed for statements that are not changeable in longer period of time and it is, again, the result from the method of Checking Update group. In this method we have variable *maxNextUpdate* that equals 28 days (a month) and it is the upper limit for which the OMS is checking all statements (the limit 28 days can be changed in the configuration file of the OMS when it will be necessary – for example the size of the data in the storage will be so huge that the OMS will have problems with checking the updates for all data in such interval). All statements of the Regular Update group have to be checked once a month. If the result of update is negative, next update will be performed in 28 days. If the result is positive, the statement is replaced and is moved to the Checking Update group with the update default schedule (again 7 update configurations), because the statement may be changing its behaviour or ontology data source started to provide more up to date information about this statement.

It is worth to emphasize that computer program decides which statements are time sensitive and which are not. It is a result of using method of the Checking Update group, without human interaction.

Figure 4.3 presents sequence diagram of interaction betweens agents during the update process. *UpdateAgent* is requesting from the *DBAgent* statements that have to be updated. Statements are grouped in data models; each model corresponds to specific ontology data sources. After receiving models from the *DBAgent*, the *UpdateAgent* for every model creates

SearchAgents and sends to them the specific model and the URL of the ontology source. *SearchAgents* request actual information about models from Ontology Providers (e.g. WrapperAgents, agents that belongs to Content Collection Subsystem).

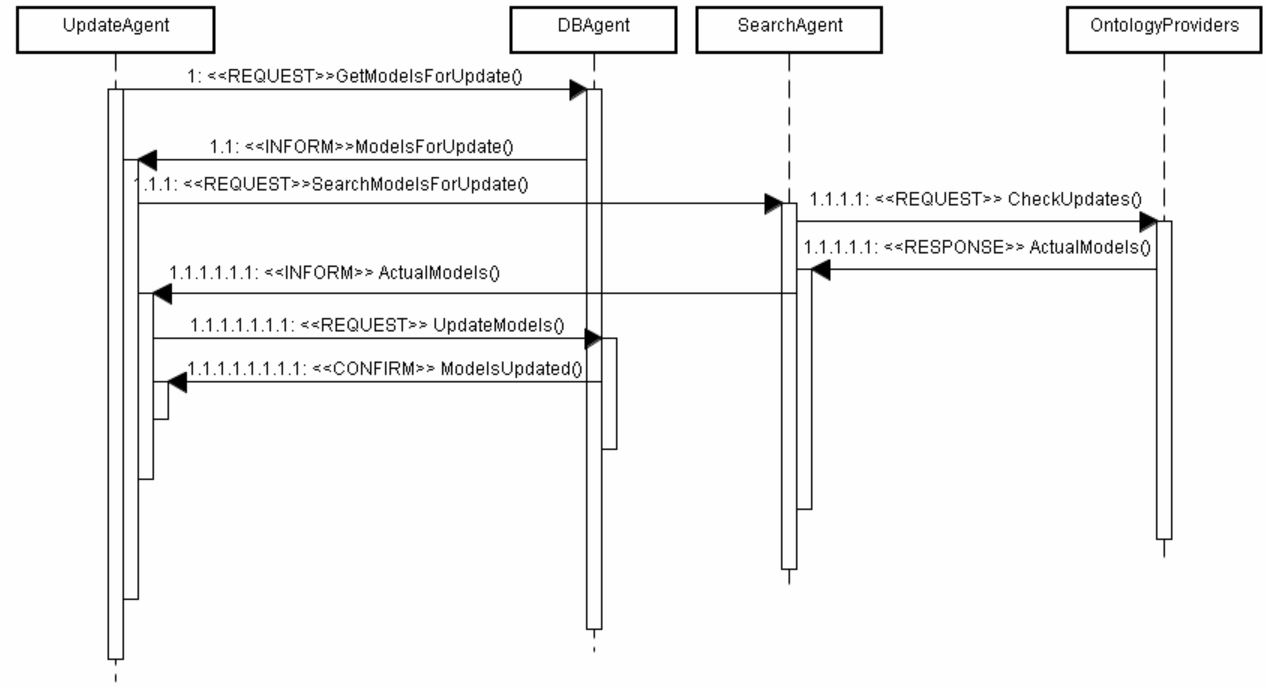

Figure 4.3 Sequence diagram – updates of information.

Then actual models are returned to the *UpdateAgent* via the *SearchAgent* and the *UpdateAgent* informs the *DBAgent* about actual models. *DBAgent* checks if statements changed their values by invoking methods of Checking Updates, Known Updates or Regular Updates.

4.3.2 New data

Agents in the Content Collection Subsystem of the TSS extract new information from ontology data sources. This information is delivered to the *NewDataAgent* of the OMS, which sends it to the *DBAgent* with the URL of the ontology source. The *DBAgent* manages all aspects of the new data, it checks if data already exists in the storage, checks the consistent and the conflicts. We can distinguish three scenarios:

1. New data does not exist in the storage.
2. New data exists in the storage and they are equal.

3. New data (properties for a specific restaurant) exists in the storage and they are not equal.

In the first scenario *DBAgent* performs the consistency checks and saves new data. In the second, only the additional information about the existing data is saved, e.g. particular statement is also in the new ontology source (URL is added to the property *hasPositiveOntSrc* of *ExtInfo* model to the specific reified statement). In the third scenario *DBAgent* additionally deals with conflicts.

Consistency checking

In section 2.2.3 *OWL Description Logics* the knowledge base that includes T-Box and A-Box has been described. The TSS includes the ontology domain of restaurants. The T-Box about this domain is in the OWL DL language. The A-Box is an instance of a particular restaurant with specific properties and values. The ontology reasoner is checking if the new received statement (or a new restaurant with all properties) is consistent with respect to given restaurant and its properties. Consistency checking is very important, because besides the fact that the data in the storage has to be up to date, it is also important it had a high quality – the data must be consistent

As the ontology reasoner is used Pellet (described in section 2.2.5 *Ontology tools*). The A-Box is created for every process of consistency checking with only one instance of the restaurant that a new statement describes. If a property of the new statement already exists it is replaced by the new value, if not, it is simply added. The instance of the Pellet reasoner is initiated with the T-Box of the restaurant ontology and the created A-Box. Pellet reasoner checks if a new A-Box is consistent with respect to the T-Box. Pellet will return *true* value if a new A-Box is consistent, otherwise *false*. If it is consistent, the next points are conflicts resolving and the new data saving.

Conflicts resolving

Conflicts between newly received data and already existing one will appear when new data from one ontology source includes values which are not equal to stored values for a particular restaurant from another ontology source, e.g. new *streetAddress* property is different for the *Restaurant "El Popo"*. Conflicts are resolved by computing the product of the *hasCF* value from the *OntologySource* model for particular ontology source and the *hasCF* value from the *ExtInfo* model for the particular statement. The winner of the conflict is the statement that has greater product of *hasCF* values.

The formula of the product is simple:

$$productCF = hasCFa * hasCFb$$

where *hasCFa* is the value of the ontology source and *hasCFb* is the value of the statement. We have to consider the case when we have a new ontology source and a new statement – the problem so called *cold start*. In such case *hasCF* value for a new ontology source and a new statement is equal 0.0 ≡ unknown, because we do not know if they are reliable. The formulas of *hasCF* values presented in the section 4.1 *OMS ontologies* are computed when the number of statements from the particular ontology source is greater than 1, also when for the particular statement the properties *hasPositiveOntSrc* or *hasNegativeOntSrc* has some values.

The product of *hasCF* values allows us to solve problems concerning local and global credibility. Specifically, assume that in the storage there is a statement with the property Prop1 and with value Val1 for the restaurant Rest1, which was delivered from the ontology source OntSrc1:

Triple(Rest1, Prop1, Val1) = Stmt1, from OntSrc1

At the same time, the *NewDataAgent* sent to the DBAgent a new statement:

Triple(Rest1, Prop1, Val2) = Stmt2, from OntSrc2

OntSrc1 and OntSrc2 already are in the storage. The value *hasCF* of OntSrc1 is 0.9, so it is very reliable ontology source in global meaning; *hasCF* of OntSrc2 is 0.2, so it is much less reliable. But for the Stmt1 the *hasCF* value is -0.5 (probably false/not reliable), so it is very unreliable statement. In other words, the OntSrc1 is very reliable globally, but it has locally less reliable statements. The *hasCF* value of the statement Stmt2 is 0.0 (unknown), because it is a new statement. After computing products of the two statements (OntSrc1.hasCF*Stmt1.hasCF=-0.45; OntSrc2.hasCF*Stmt2.hasCF=0.0) Stmt2 wins, it has greater *productCF* value. In the case when both products will be equal, the conflict wins the statement for which the ontology source has grater value of the *hasCorrectTripples* property. Furthermore, if the values of the *hasCorrectTripples* property will be equal, the conflict wins the statement that was already in the storage.

The special case when resolving conflicts is if the property is *Functional* and it is a literal with a text value. In this case a new received value can be subtly different from the stored one, e.g. the streets addresses: ‘ul Jana Niepodleglosci’, ‘ul. Jana Niepodległości’ (the polish letters or the punctuations marks can be often omitted in the ontology sources). Both values can be recognized as equivalent. But computer program is checking theses values

single character by single character to check whether they are equal. To solve this potential problem the Levenshtein distance algorithm (bottom-up dynamic programming algorithm) is used. The Levenshtein distance algorithm between two strings returns the number of minimal operations needed to transform one string to another. These operations are insertion, deletion and substitution of single characters. In the given example Levenshtein distance is equal 3. Values are treated as equivalent when Levenshtein distance is smaller or equal 3 (value 3 is not proved as a faultless constant, and therefore it can be changed in the configuration file of OMS).

Data saving

Process of saving new data is quite complex. Even the process of inserting one new statement to the *Restaurant* model requires few additional inserts or updates. First, the *DBAgent* is checking if the ontology source of the new statement is already in the storage. If it is not, it has to add new instance to the *OntologySource* model of this source. If it is, it has to update properties *hasCorrectTripples* or *hasWrongTripples*. Second, a new statement is added to the *Restaurant* model, and is reified and a new instance of the *ExtInfo* model is created with primary update configuration value set to the *hasUpdateDateTime* property and the URI of the ontology source instance set to the *hasMainOntSrc* property. Properties like *hasPositiveOntSrc* or *hasNegativeOntSrc* are also updated for the particular statement when the *NewDataAgent* receives new information, and when values are equal, then the property *hasPositiveOntSrc* is added to the ontology source for a given statement, otherwise to the property *hasNegativeOntSrc* if a given statement will win the conflict. If it will loose, the old statement is deleted and a new one added. All steps of inserting process are performed.

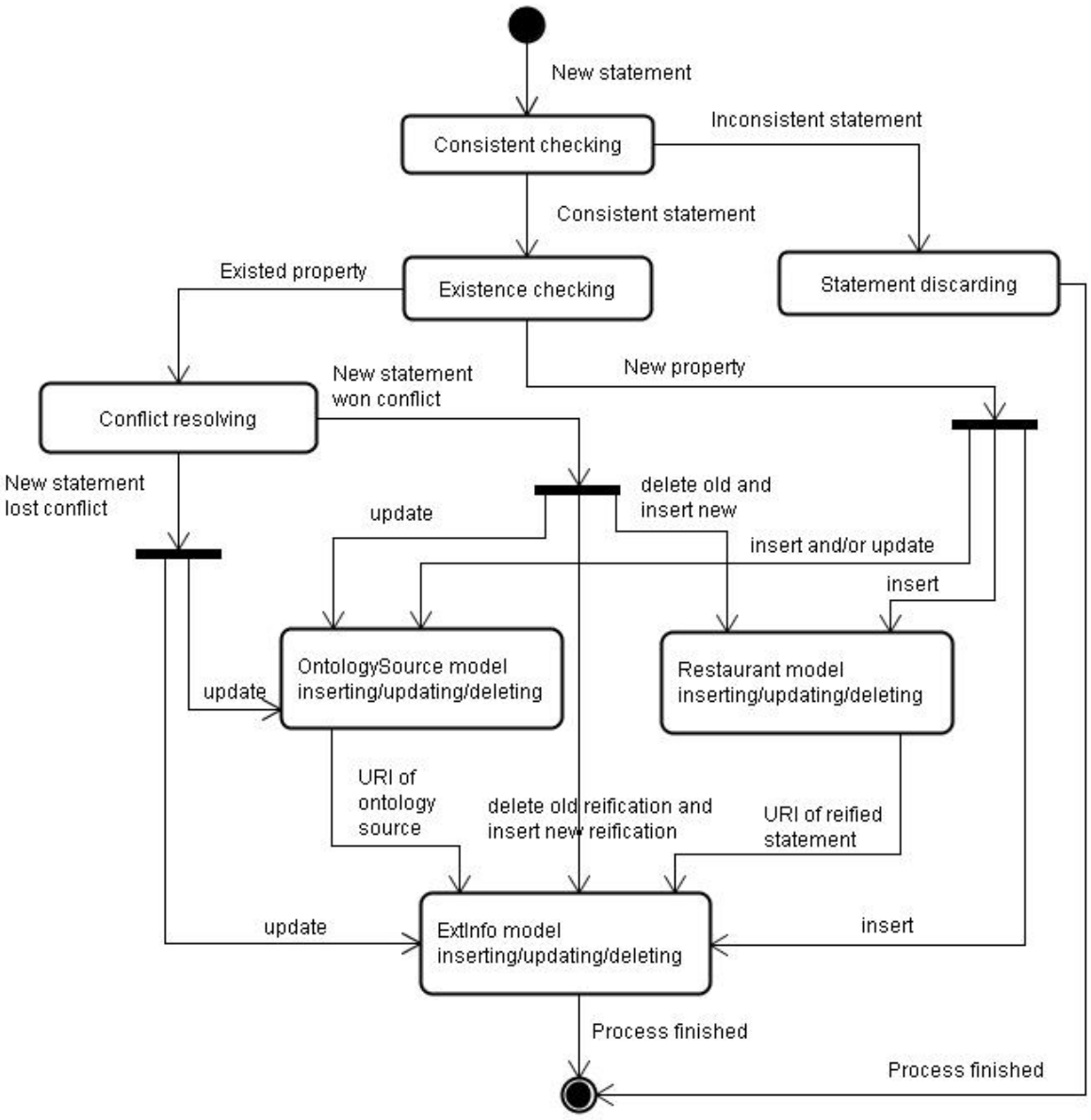

Figure 4.4 The activity diagram of new data checking.

Figure 4.4 presents activity diagram of process when new data is received by the *DBAgent*.

4.3.3 Incomplete data

Together with actual and reliable data in the storage we have to focus on completeness of that data. The ontology domain of restaurants in the TSS is very complex; it includes many properties like *phone*, *reservationURL*, *capacity* etc. One particular restaurant is treated as complete when it has values of all properties of the restaurant ontology. It is treated as

incomplete if it is missing value for at least one property. The incomplete restaurants are demarcated by the property *hasIncompleteCheckDateTime* in the model *ExtInfo*.

The property *hasIncompleteCheckDateTime* has the form of date and time for checking incomplete restaurant and one parameter that is value to next update. This parameter is bound with minimum value 1 and maximum 7 (incomplete data is checked at most once a day or at least once a week). Default value of this parameter is equal to minimum value times 2.

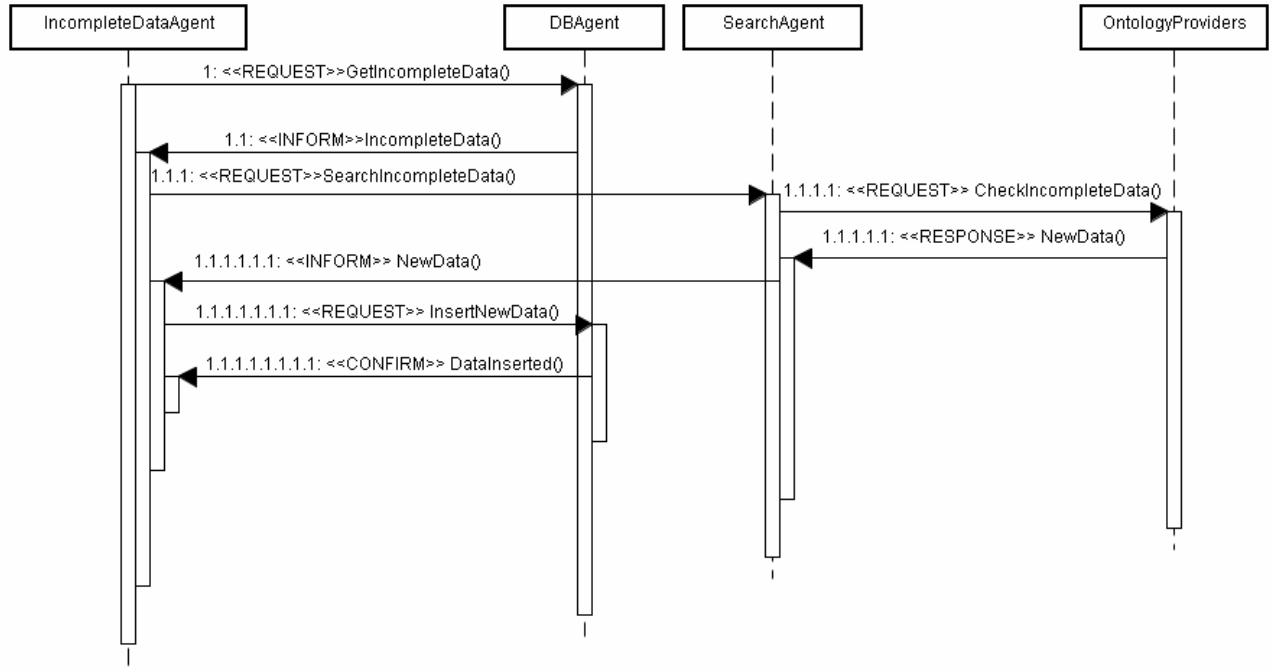

Figure 4.5 Sequence diagram – completeness of data checking and requesting.

Agent *IncompleteDataAgent* is requesting from *DBAgent* to check date and time of the *hasIncompleteCheckDateTime* property (figure 4.5). *DBAgent* responses and *IncompleteDataAgent* is sending a request to the *SearchAgents* to search in ontology sources for the missing data.

If the *SearchAgents* will not find the data then the property *hasIncompleteCheckDateTime* is set with new value according to the formula:

*hasIncompleteCheckDateTime = (date and time)+ nextUpdate | nextUpdate*2*

So next checking will be performed after two days. If *SearchAgents* finds any missing data then the property becomes:

hasIncompleteCheckDateTime = (date and time)+ nextUpdate/2 | nextUpdate/2

next date and time of checking is decreased twice. The parameter *nextUpdate* is computed with respect to the minimum and the maximum value which can be modified in the configuration file of the OMS. New found data is saved to the storage with respect to all requirements of new data saving in the OMS.

4.3 OMS working outlook

This section presents the working outlook of the OMS. Two cases are presented:

1. Saving new data – one restaurant.
2. Perform updates of the new restaurant.

First case – saving new data

NewDataAgent received from the Ontology provider *http://chefmoz.org/* the new restaurant "Paradios" from the city Kawice, which is ontology demarcated (RDF/XML syntax):

```
<res:Restaurant rdf:ID="Poland_DS_Kawice_Paradiso__Restauracja1011064378">
   <loc:streetAddress rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
   >Kawice 54a, (p-ta Prochowice) </loc:streetAddress>
   <loc:state rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
   >DS</loc:state>
   <loc:zip rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
   >59-230</loc:zip>
   <res:title rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
   >Paradiso, Restauracja</res:title>
   <loc:phone rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
   >+48 (76) 858 46 82 </loc:phone>
   <res:locationPath rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
```

```
    >Poland/DS/Kawice</res:locationPath>
    <loc:city rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Kawice</loc:city>
    <loc:country rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Poland</loc:country>
</res:Restaurant>
```

The ontology data of the new restaurant is sent to the *DBAgent* that performs *Consistency checking*, *Conflicts resolving* and *Data saving*. First the ontology reasoner checks if the data of the new restaurant is consistent with respect to the ontology domain of the restaurants (the T-Box). The output of the ontology reasoner (*Pellet*) is shown on listing 4.2.

```
INFO [main] (KnowledgeBase.java:1474) - Expressivity: ALF(D), Classes: 18 Properties: 96 Individuals: 408 Strategy: SHNStrategy
INFO [main] (ABox.java:1529) - ABox consistency for 0 individuals
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 452 Time: 32
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Restaurant#AccessibilityCode for 0 individuals []
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 0
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Restaurant#RestaurantCategoryCode for 0 individuals []
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 0
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Location#IndexPointCode for 0 individuals []
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 0
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Location#LocationCategoryCode for 0 individuals []
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 0
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Restaurant#ReservationCode for 0 individuals []
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 0
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Restaurant#RestaurantServiceInfo for 0 individuals []
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 0
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Restaurant#Restaurant for 0 individuals []
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 16
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Restaurant#AlcoholCode for 0 individuals []
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 0
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Money#MeanOfPayment for 0 individuals []
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 0
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Restaurant#DressCode for 0 individuals []
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 0
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Restaurant#DinerReview for 0 individuals []
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 0
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Restaurant#SmokingCode for 0 individuals []
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 0
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Restaurant#FeatureCode for 0 individuals []
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 0
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Restaurant#ParkingCode for 0 individuals []
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 0
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Location#AttractionCategoryCode for 0 individuals []
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 0
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Location#Location for 0 individuals []
```

```
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 0
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Money#FuzzyPriceCode for 0 individuals []
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 0
INFO [main] (ABox.java:1531) - Consistency http://www.ibspan.waw.pl/tss/Restaurant#CuisineCode for 0 individuals []
INFO [main] (ABox.java:1618) - Consistent: true Tree depth: 1 Tree size: 1 Time: 0
Consistent=true
```

Listing 4.2 The ontology reasoner output (*Pellet*)

The data of the new restaurant is consistent (Consistent=true – listing 4.2). Next step is *Conflicts resolving*. The *DBAgent* checks if the new data is in the storage; it is not and the *Conflicts resolving* is not performed – the *DBAgent* treats the new data as a reliable data. Now the *DBAgent* can save the new data into the storage. Every statement is reified and to every reified statement is set URI, e.g. for the statement:

<loc:phone rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
>+48 (76) 858 46 82 </loc:phone>

is set the URI:

http://www.ibspan.waw.pl/tss/db/ExtInfo#ReifStmt_2886294508_1209391742453

where:

- http://www.ibspan.waw.pl/tss/db/ExtInfo is namespace for the *ExtInfo* model
- ReifStmt_2886294508_1209391742453 is local name of the URI constructed as follows: the string "ReifStmt_" + result of the CRC32[37] checksum of the string of the given statement + "_" + (Java System.currentTimeMillis())

Reified statements are set to the *ExtInfo* model with the RDF/XML listing below:

```
<oms:ExtInfo rdf:ID="ReifStmt_2886294508_1209391742453">
  <oms:hasUpdateDateTime rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >2008-03-11 16:09:02;1,1,7,0&2008-03-12 16:09:02;1,1,7,0&2008-03-13
  16:09:02;1,1,7,0&2008-03-14 16:09:02;1,1,7,0&2008-03-15 16:09:02;1,1,7,0&
  2008-03-16 16:09:02;1,1,7,0&2008-03-17 16:09:02;1,1,7,0&
  </oms:hasUpdateDateTime>
<oms:hasIncompleteCheckDateTime rdf:datatype="http://www.w3.org/2001/
  XMLSchema#string">2008-03-11 16:09:02;1</oms:hasIncompleteCheckDateTime>
  <oms:hasCF rdf:datatype="http://www.w3.org/2001/XMLSchema#float"
```

[37] CRC32 - Cyclic Redundancy Code checksum with 32 bits the polynomial exponent

```
  >1.0</oms:hasCF>
  <oms:hasMainOntSrc rdf:resource=" http://www.ibspan.waw.pl/tss/
        db/OntologySource#http1000chefmoz1110com"/>
</oms:ExtInfo>
```

```
/**
 * Add properties to ExtInfo model for given reified statement
 * @param mExtInfoDB reference to the ExtInfo model in persistent storage of the JENA
 * @param reifStmt URI of the reified statemet
 * @param rOntSrc resource of the main ontology source
 */
public void addPrimaryExtInfoToReifStmt(OntModel mExtInfoDB, String reifStmt, Resource rOntSrc) {
  try {
      // begin transaction
      mExtInfoDB.begin();
      // create ExtInfo resource
      ExtInfo extInfo = OMSFactory.createExtInfo(mExtInfoDB.createResource(reifStmt), mExtInfoDB);
      // set update date and time property
      extInfo.setHasUpdateDateTime(
          getUpdateCheckConf(CHECK_UPDATE,MIN_SINCE_LAST_UPDATE,
          MIN_NEXT_UPDATE,ZERO_ITERATION));
      // set incomplete check date and time property
      extInfo.setIncompleteCheckDateTime(
          getIncompleteCheckConf(CHECK_INCOMPLETE,MIN_SINCE_LAST_UPDATE,NEXT_UPDATE));
      // set Certaitny Factor value
      extInfo.setHasCF(0.0f);
      // set resource of the main ontology source
      extInfo.setHasMainOntSrc(rOntSrc);
      // commit transaction
      mExtInfoDB.commit();
  } catch (JastorException e) {
      // abort transaction
      mExtInfoDB.abort();
       throw new RuntimeException("Cannot add extInfo to the reified statement.", e);
  }
}
```

Listening 4.3 Method for set properties to the *ExtInfo* model for the reified statement (JENA API in Java language)

The listening code 4.3 provides the Java method to set properties to the ExtInfo model by using the JENA API (ontological database described in section 2.2.5 *Ontology tools*).

Second case – updates of the data

UpdateAgent is requesting from the *DBAgent* statements that should be updated according to the property *hasUpdateDateTime*. The *DBAgent* is querying the data in the storage using the SPARQL language (listing 4.4).

```
SELECT ?x ?y ?z
WHERE
{ ?x <http://www.ibspan.waw.pl/tss/OMS#hasUpdateDateTime> ?y .
  ?x <http://www.ibspan.waw.pl/tss/OMS#hasMainOntSrc> ?z . }
```

Listening 4.4 SPARQL query – get statements for update

In listing 4.3 the variable ?x corresponds to the resource of the reified statement, the variable ?y is the value of the *hasUpdateDateTime* property, the variable ?z is the resource of the main ontology source. The variable ?y is checked if statement should be updated, if it is then the process of update is performed (figure 4.3). Figure 4.6 provides the updates process visualized by the *Sniffer Agent* (chapter 2.3.2 *Agent platform – JADE*) in the JADE platform.

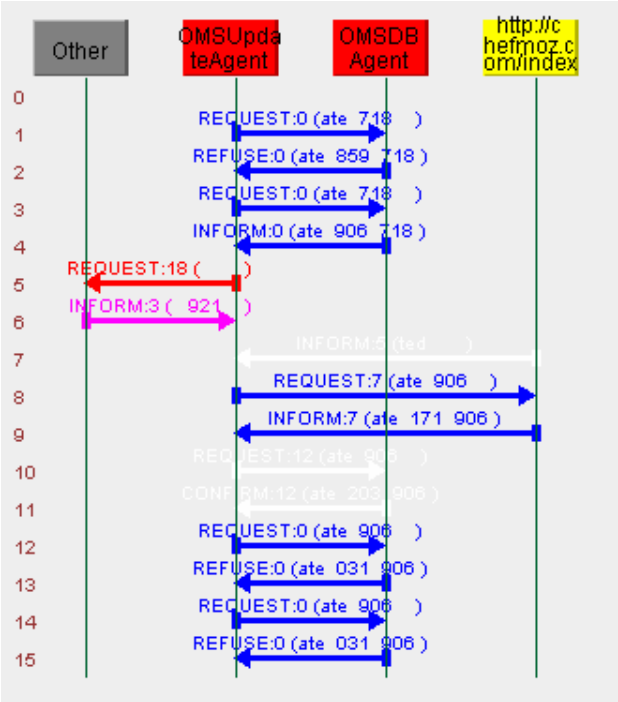

Figure 4.6 Sequence diagram – updates of information – visualized by the *Sniffer Agent* (JADE)

In figure 4.6 in line 1 *UpdateAgent* has requested from *DBAgent* statements that should be updated. At the timestamp of the request there was nothing to update, so the *DBAgent* refused the request (line 2). After one minute UpdateAgent requested again statements from the DBAgent (line 3). This time there existed statements that should have been updated and the *DBAgent* sent them to the *UpdateAgent* (line 4). *UpdateAgent* requested initialization of *SearchAgent* (lines 5, 6) and created *SearchAgent* (yellow box) inform from the *UpdateAgent* that it was successfully initialized (line 7). Then the *UpdateAgent* requested that the *SearchAgent* get new values of statements from the Ontology providers (line 8). The *SearchAgent* sent new values to the *UpdateAgent* (line 9) which requested that the *DBAgent* performs the update process (line 10). The *DBAgent* confirmed the update process (line 11) and the *UpdateAgent* again requested from the *DBAgent* statements that should be updated (lines 12 – 15).

4.4 OMS implementation

Agents in the OMS were implemented in JADE 3.4.1. All ontology data manipulations were achieved by using JENA 2.4, data was stored in the PostgreSQL[38] 8.0 database. JENA connected with the PostgreSQL database via the JDBC[39]. Ontology reasoner was Pellet 1.5.1, which interacted with JENA. Two ontology models: *OntologySource* and *ExtInfo*, were created by using Protégé 3.2.1.

Listing 4.5 provides the configuration file of the OMS.

```
# Agent-based Travel Support System
# Ontology Management System configuration file
# ------ Data updates ------
omsZeroIteration=0
omsFiniteIteration=3
omsMinSinceLastUpdate=1
omsMinNextUpdate=7
omsMaxNextUpdate=28
# ------ Incomplete data ------
omsMinIncompleteCheck=1
```

[38] http://www.postgresql.org/
[39] Java DataBase Connectivity

```
omsMaxIncompleteCheck=7
# ------ Levenshtein distance ------
omsLevenshteinDistance=3
# ------ UpdateAgent ------
omsUAFrequencyChecking=10000
omsUAMaxCntStatements=1000
# ------ NewDataAgent ------
omsNDAFrequencyChecking=10000
# ------ IncompleteDataAgent ------
omsIDAFrequencyChecking=10000
omsIDAMaxCntStatements=1000
```

Listening 4.5 Configuration file of the OMS

Agent properties like *omsUAFrequencyChecking*, *omsNDAFrequencyChecking*, *omsIDAFrequencyChecking* inform how often particular agents request data from the *DBAgent*, e.g. here, *UpdateAgent* data to be updated has value=10000, meaning that the *UpdateAgent* is requesting data to be updated every 10 seconds. Properties like *omsUAMaxCntStatements* and *omsIDAMaxCntStatements* inform how many statements can be returned from the SPARQL query, e.g. the *IncompleteDataAgent* can check in one-shot the specific amount of statements that are missing (value=1000 – the maximum number of missing statements to check in one-shot). Otherwise, with not bounded number of missing statements, SPARQL query could return so large number of statements that the *IncompleteDataAgent* would not be able to perform its task – e.g. the *java.Lang.OutOfMemory exception* could have been thrown. All values of the properties in the configuration file can be changed for tuning and optimizing the OMS.

5. Conclusions

In this Master thesis I designed and implemented the Ontology Management System that performs functions of the Content Management Subsystem that is a part of the Travel Support System. The Ontology Management System was implemented by using the Semantic Web and the software agents technologies. It is mainly responsible for managing the data in the Content Storage of the TSS – its goal is that data has to be up to date, reliable and complete.

The OMS includes several agents that form a Multi-agent System and that co-operate which each other to meet requirements related to functionalities of the CMS. Agents are designed according to the *decomposition* methodology and are implemented in JADE. Two additional ontology models were defined: *OntologySource* and *ExtInfo*. All data in the OMS is ontologically demarcated in OWL language. Functionalities of the OMS are as follows: *Data updates* – are divided in three groups: Checking Updates (algorithm to classify data to time sensitive or not), Known Updates (updates of time sensitive data) and Regular Updates (updates of no time sensitive data); *New data* - Consistency checking (with the Pellet ontology reasoner), Conflicts resolving (by calculating Certainty Factors) and Data saving; and *Incomplete data* (data incompleteness checking and requesting missing information). All data manipulations are performed by using JENA API. Working status of the OMS can be tuned by changing the constant values in the configuration file of the OMS to achieve more efficient results of *Data updates*, *New data* processes or *Incomplete data* checking.

The probable future technical restriction of the OMS would be the problem to perform functions like *Data updates*. This problem can appear when there will be a very big number of restaurants in the storage. The technical restrictions like speed of the Internet connection or time of querying the database could become bottlenecks of the OMS. In this situation the administrator of the OMS will have to change values in the configuration file of the OMS – like the upper limit of checking Known updates; e.g. from 28 days to 56 days. As the future work can be extension to the OMS that values in the configuration file will be changing automatically as the result of any solution by using e.g. neural networks or data mining – the tune of the OMS will be achieved without human interaction.

At this stage of development of the Internet, there are no actual VCP sources. In my opinion, except of looking VCPs on the Internet, the TSS has big chance to become such a Verified Content Provided source itself.

6. References

1. Agent Communication Language (ACL) Specification. http://www.fipa.org/repository/aclspecs.html

2. AgentLab - Travel Support Project. http://agentlab.swps.edu.pl/agents_TSS.html

3. AgentLab - Travel Support Project. Codes and ontologies. http://agentlab.swps.edu.pl/agent_papers/tss-1.0-all.zip

4. Arpinar, Budak. Giriloganathan, Karthikeyan. Aleman-Meza, Boanerges. 2006. Ontology Quality by Detection of Conflicts in Metadata. LSDIS Lab, Computer Science Dept. University of Georgia. http://km.aifb.uni-karlsruhe.de/ws/eon2006/eon2006arpinaretal.pdf

5. Bellifemine, Fabio. Caire, Giovanni. Greenwood, Dominic. 2007. Developing Multi-Agent Systems with JADE. John Wiley & Sons Ltd, England.

6. Berners-Lee, Tim. 1998. What the Semantic Web can represent. http://www.w3.org/DesignIssues/RDFnot.html

7. Berners-Lee, Tim. Fischetti, Mark. 1999. Weaving the Web. Harper Collins Publishers.

8. Berners-Lee, Tim. 2000. Semantic Web – XML2000. http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html.

9. Berners-Lee, Tim. Hendler, J. Lassila, O. 2001. The Semantic Web, Scientific American.

10. Booch, Grady. 1994. Object-oriented analysis and design with applications (2nd ed.). Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc.

11. Caire, Giovanni. 2003. Jade Tutorial: Jade Programming for Beginners. TILab.

12. Caire, Giovanni. 2004. Jade Tutorial: Application-defined Content Langu-ages and Ontologies. TILab.

13. Chavarkar, Pradnya. Agent Oriented Programming. Seminar Report. Department of Computer Science and Engineering Indian Institute of Technology, Bombay Mumbai.

14. ChefMoz. 2005.ChefMoz dining guide: http://chefmoz.org/

15. Chmiel, Krzysztof. Tomiak, Dominik. Gawinecki, Maciej. Karczmarek,Paweł. Szymczak, Michał. Paprzycki, Marcin. 2004. Testing the Efficiency of JADE Agent Platform. Strony 49–56 z: ISPDC ’04: Proceedings of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (ISPDC/HeteroPar’04). Washington, DC, USA: IEEE Computer Society.

16. McAllister, David. Massachusetts Institute of Technology.http://www.rattlesnake.com/ notions/certainty-factors.html

17. Daconta, Michael C.. Obrst, Leo J. Smith, Kevin T. 2003. The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management. Indianapolis, Indiana, USA: Wiley Publishing, Inc.

18. Davies, John. Fensel, Dieter. Harmelen, Frank. 2003. Towards the Semantic Web: Ontology-Driven Knowledge Management. John Wiley & Sons.

19. Esperonto Project IST-2001-34373. http://www.esperonto.net/semanticportal/jsp/frames.jsp

20. FIPA Request Interaction Protocol. http://www.fipa.org/specs/ fipa00026/SC00026H.pdf

21. Gawinecki, Maciej. 2005. Modelowanie użytkownika na podstawie interakcji z systemem opartym o technologie WWW. Master Thesis. Wydział Matematyki i Informatyki, Uniwersytet Adama Mickiewicza, Poznań.

22. Gawinecki, Maciej. Gordon, Minor. Paprzycki, Marcin. Szymczak, Michał. Vetulani, Zygmunt. Wright, Jimmy. 2005a. Enabling Semantic Referencing of Selected Travel Related Resources. Pages 271–288 from: Abramowicz, Witold. BIS 2005: Proceedings of the 8th International Conference on Business Information Systems. Poznań University of Economics Press.

23. Gawinecki, Maciej. Gordon, Minor. Nguyen, Ngoc Thanh. Paprzycki, Marcin. Szymczak, Michał. 2005b. RDF Demarcated Resources in an Agent Based Travel Support System. Pages 271–288 from: Proceedings of the 17th Mountain Conference of the Polish Information Society.

24. Gawinecki, Maciej. Kruszyk, Mateusz. Paprzycki, Marcin. 2005. Ontology-based Stereotyping in a Travel Support System. Proceedings of the XXI Fall Meeting of Polish Information Processing Society. PTI Pres.

25. Gawinecki, Maciej. Kruszyk, Mateusz. Paprzycki, Marcin. Ganzha, Maria. 2007. Pitfalls of agent system development of the basis of a Ravel Support System. Polish academy of Sciences. Systems Research Institute. Warsaw, Poland.

26. Gordon, Minor. Paprzycki, Marcin. 2005. Designing Agent Based Travel Support System. pages 207–214 z: Proceedings of the ISPDC 2005 Conference. Los Alamitos, CA, USA: IEEE Computer Society Press.

27. Gasiorowski, Rafal. 2006. Using Ontologies to Organize Data Collected from the Internet. Master Thesis. Warsaw University of Technology.

28. Gruber, TR. 1992. A translation approach to portable ontologies. Knowledge Acquisition 5(2):199–220, 1993. http://ksl-web.stanford.edu/KSL_Abstracts/KSL-92-71.html

29. Hendler, James A. 1999. Is There an Intelligent Agent in Your Future? Nature Webmatters.

30. JADE. Java Agent DEvelopment framework. http://jade.tilab.com/

31. Java. http://java.sun.com/

32. JENA, Ontology database. http://jena.sourceforge.net/

33. Jennings, Nicholas R. 1999. Agent-oriented software engineering. Proceedings of the 12th international conference on Industrial and engineering applications of artificial intelligence and expert systems : multiple approaches to intelligent systems. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

34. Jennings, Nicholas R. 2001. An agent-based approach for building complex software systems. CACM 44, 4. Pages 35-41.

35. Kaczmarek, Paweł. 2005. Multimodalna komunikacja agentów programowych z użytkownikiem. Master Thesis. Wydział Matematyki i Informatyki, Uniwersytet Adama Mickiewicza, Poznań.

36. Lutz, Carsten. Sattler, Ulrike. Description Logics Course. http://lat.inf.tudresden.de/~clu/esslli.html, Day 1

37. Maes, P. 1994. Agents that Reduce Work and Information Overload. Communications of the ACM.

38. Muthukumaraswamy, Karthik Chinnayan. 2005. Supplying Data for an RDF-based Content Management System.

39. Nowak, Marcin. 2004. Pajęczyna II. CHIP, 7. http://www.chip.pl/arts/archiwum/n/articlear_107066.html

40. N3. 2005. Notation 3 – Ideas about Web architecture. http://www.w3.org/DesignIssues/Notation3.html

41. Ontology Web Language Features. http://www.w3.org/TR/owl-features/

42. Paprzycki, Marcin. 2003. Agenci programowi jako metodologia tworzenia oprogramowania. Z. Huzar, Z. Mazur (ed.), Problemy i Metody Inżynierii Oprogramowania, WNT, Warszawa.

43. Paprzycki, Marcin, Gilbert, Austin. Nauli, Andy. Gordon, Minor. Williams, Steve.Wright, Jimmy. 2004. Indexing agent gathered data in an e-travel system, Informatica.

44. Pellet, Ontology reasoner. http://pellet.owldl.com/

45. Pisarek, Szymon. 2005. Ontologically Oriented Internet Search. Master Thesis. Warsaw University of Technology.

46. PostgreSQL, the relational database. http://www.postgresql.org/

47. Protégé, Ontology editor. http://protege.stanford.edu/

48. RDF. 2005. Resource Description Framework (RDF). http://www.w3.org/RDF

49. RDF/XML. 2005. RDF/XML Syntax Specification. http://www.w3.org/TR/rdf-syntax-grammar/

50. Russell, Stuart. Norvig, Peter. 2002. Artificial Intelligence: A Modern Approach. Prentice Hall.

51. Semantic Web on XML, slide Architecture. http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html

52. SPARQL Specification. http://www.w3.org/TR/rdf-sparql-query/

53. Szymczak, Michał. 2006. Modeling Using Ontological Technologies. Master Thesis. Wydział Matematyki i Informatyki, Uniwersytet Adama Mickiewicza, Poznań.

54. Wikipedia: Ontology. Computer Science context: http://pl.wikipedia.org/wiki/Ontologia

55. Wikipedia: Ontology. http://en.wikipedia.org/wiki/Ontology

56. Wooldridge, M. 1997. Agent-based software engineering. IEE Proceedings Software Engineering.