ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea in Ingegneria e Scienze Informatiche Magistrale

SEMANTIC MATCHMAKING ON RESOURCE-CONSTRAINED DEVICES

Corso di: Sistemi Distribuiti

Elaborato di:
MISSERE MATTIA

Docente:
Prof. ANDREA OMICINI

ANNO ACCADEMICO 2015-2016

PAROLE CHIAVE

Sematic Web of Things

Reasoners

Web Ontology Language OWL

Mini-ME

Protégé

Indice

Related work

My work is to study the scope of semantic matchmaking, then is to analyze and test Mini-ME and search how it may differ from other reasoners such as Fact++, HermiT and Pellet.

Capitolo 1

General overview

1.1 Internet of Things

The **Internet of Things** (**IoT**) [8] is a system of interrelated computing devices, mechanical and digital machines, objects, animals or people that are provided with unique identifiers and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction.
The interconnection of data developed by the large and heterogeneous amount of related objects is the main element that marks the passage from what is the current Internet of Things to what will be the Internet of Things of tomorrow. In fact, until the things don't have the ability to communicate and exploit the data of all the other things around them intelligently, they will remain only simple electronic devices while having high connectivity capabilities.
This goal will not be easy to achieve, just because the things that surround us are highly heterogeneous and not always ready to share information.
To solve these problems, the researchers are working on projects such as *Semantic Web* and *Open Data*, which seek to find a solution for building (by using the data in a format suitable for querying, interpretation and automatic processing) and for the sharing of knowledge.

1.2 Semantic Web

The **Semantic Web** (**SW**) [21] is an extension of the existing *World Wide Web*. It provides a standardized way of expressing the relationships between web pages, to allow machines to understand the meaning of hyperlinked information. It is based on machine-readable information and builds on XML technology's capability to define customized tagging schemes and RDF's (Resource Description Framework) flexible approach to representing data. The Semantic Web provides common formats for the interchange of data (where on the Web there is only an interchange of documents).
It also provides a common language for recording how data relates to real world objects, allowing a person or a machine to start off in one database, and then move through an unending set of databases which are connected not by wires but by being about the same thing.

1.3 Semantic Web of Things

The Semantic Web and Internet of Things visions are converging toward the so called **Semantic Web of Things** (**SWoT**) [22]. It aims to enable smart semantic-enabled applications and services in Ubiquitous contexts. Due to architectural and performance issues, it is currently impractical to use existing Semantic Web reasoners. They are resource consuming and are basically optimized for standard inference tasks on large ontologies. On the contrary, SWoT use cases generally require quick decision support through semantic matchmaking in resource-constrained environments. For this reason, special Reasoners have been developed that can do semantic matchmaking even in limited resources such as mobile, it is the example of Mini-Me.

1.4 Ontology and OWL

Ontologies are a formal way to describe taxonomies and classification networks, essentially defining the structure of knowledge for various domains: the nouns representing classes of objects and the verbs representing relations between the objects. The ontologies are meant to represent information on the Internet and are expected to be evolving almost constantly. Infact the

ontologies are typically flexible as they are meant to represent information on the Internet coming from all sorts of heterogeneous data sources. The **Web Ontology Language** (**OWL**) is a family of knowledge representation languages for authoring ontologies.

1.5 Reasoner

A **Reasoner** is a key component for working with OWL ontologies. In fact, virtually all querying of an OWL ontology (and its imports closure) should be done using a reasoner. This is because knowledge in an ontology might not be explicit and a reasoner is required to deduce implicit knowledge so that the correct query results are obtained.

1.6 OWL API

The **OWL API** [15] is a Java API and reference implementation for creating, manipulating and serialising OWL Ontologies. The latest version of the API is focused towards OWL 2.
The OWL API includes:

- An API for OWL 2 and an efficient in-memory reference implementation;
- RDF/XML parser and writer;OWL/XML parser and writer;
- OWL Functional Syntax parser and writer;
- Turtle parser and writer;
- KRSS parser;
- OBO Flat file format parser;
- Reasoner interfaces for working with reasoners such as FaCT++, HermiT, Pellet and Mini-ME.

Capitolo 2

Mini-ME

Mini-ME (the **Mini Matchmaking Engine**) [12] is a compact matchmaker and reasoner for the $\mathcal{ALN}$ (*Attributive Language* with *unqualified Number restrictions*) Description Logic (DL).
$\mathcal{AL}$ is the base language witch allows atomic negation, concept intersection, universal restrictions and limited existential quantification, while $\mathcal{N}$ characterizes the ability to define cardinality restrictions.
It is aimed to semantic matchmaking for resource/service discovery in mobile and ubiquitous contexts, although it is also a general-purpose SemanticWeb inference engine.
Mini-ME is suitable to a widespread class of applications where a large number of low-complexity component resources can be aggregated to build composed services with growing semantic complexity. This is fit for the computational and power supply limitations of resource providers in ubiquitous contexts and to their short storage availability. An "agile" service discovery architectures able to select, assemble and orchestrate on the fly many elementary components is more manageable and effective in ubiquitous applications.
Mini-ME uses the `OWL API` (`3.2.4` or more recent) [15] to parse and manipulate Knowledge Bases in all OWL 2 supported syntaxes and it uses the `Colt 1.2.0 API` [1] that they are a set of Open Source Libraries for High Performance Scientific and Technical Computing written in Java and developed at CERN.
It exploits structural inference algorithms on unfolded and *CNF* (Conjunctive Normal Form) normalized concept expressions for efficient computations

also on resource-constrained platforms.
Mini-ME implements both standard reasoning tasks for Knowledge Base (KB) management (**subsumption**, **classification**, **satisfiability**) and non-standard inference services for semanticbased resource discovery and ranking (**abduction**, **contraction**, **covering**).
It is developed in `Java`, with `Android` as the main target computing platform. Mini-ME supports the `OWLlink protocol` [11] for standard application-reasoner interaction. `OWLlink protocol` is a functional *Tell/Ask* interface exploiting HTTP as the underlying transfer protocol. Furthermore, reasoner and GUI (Graphical User Interface) plug-ins have been developed for the `Protégé` [19] ontology editor.

2.1 Support language

In DL-based reasoners, an ontology $\mathcal{T}$ (Terminological Box or TBox) is composed by a set of assertions in the form A $\subseteq$ D (inclusion) or A $\equiv$ D (definition), where A and D are concept expressions.
Particularly, a simple-TBox is an acyclic TBox such that: A is always an atomic concept; if A appears in the left hand side of a concept definition assertion, then it cannot appear also in the left hand side of any concept inclusion assertion.
Mini-ME supports the $\mathcal{ALN}$ (Attributive Language with unqualified Number restrictions) DL, which has polynomial computational complexity for standard and non-standard inferences in simple-TBoxes, whose depth of concept taxonomy is bounded by the logarithm of the number of axioms in it.
Syntax and semantics of ALN constructs and simple-TBoxes:

Symbol	Description	Name
$\top$	every individual as an instance	Top
$\bot$	empty concept	Bottom
$\sqcup$	union or disjunction of concepts	Union
$\sqcap$	intersection or conjunction of concepts	Intersection
$\sqsubseteq$	concept inclusion	Inclusion
$\neg$	negation or complement of concepts	Atomic negation
$\equiv$	concept equivalence	Equivalence

2.2 Inference services

Mini-ME exploits structural algorithms for standard and non-standard inference services on (unfolded and normalized) concept expressions. Main peculiarity is a careful optimization of the implementation of both algorithms and data structures, which enable efficient computations even on resource-constrained devices such as mobile and embedded ones.
When loading a Knowledge Base, Mini-ME performs a preprocessing in order to execute **unfolding** and **CNF normalization**.
Unfolding: given a TBox $\mathcal{T}$ and a concept C, the unfolding procedure recursively expands references to axioms in $\mathcal{T}$ within the concept expression itself. In this way, $\mathcal{T}$ is not needed any more when executing subsequent inferences.
Normalization: transforms the unfolded concept expression in CNF through a set of pre-defined substitutions, and it preserves semantic equivalence. The normal form of an unsatisfiable concept is simply $\bot$.

Standard reasoning services are supported:

- **Concept Satisfiability (Consistency)**: in a semantic matchmaking framework, given a request D and a supplied resource S as concept expressions with regard to a common TBox $\mathcal{T}$, satisfiability allows to determine whether there is a partial (disjoint) match or not, by checking whether $\mathcal{T} \vDash S \sqcap D \sqsubseteq \bot$ holds or not. Due to CNF properties, satisfiability check is trivially performed during normalization.

- **Subsumption check**: subsumption determines whether the resource is a full (subsume) match for the request or not, by checking whether $\mathcal{T} \vDash S \sqsubseteq D$ holds or not.

Non-standard reasoning services are supported:

- **Concept Contraction**: given a request D and a supplied resource S, if they are not compatible with each other, Contraction determines which part of D is conflicting with S.

- **Concept Abduction**: whenever D and S are compatible, but S does not imply D, Abduction allows to determine what should be hypothesized in S in order to completely satisfy D also enabling a logic-based relevance ranking of a resource with regard to a given request.

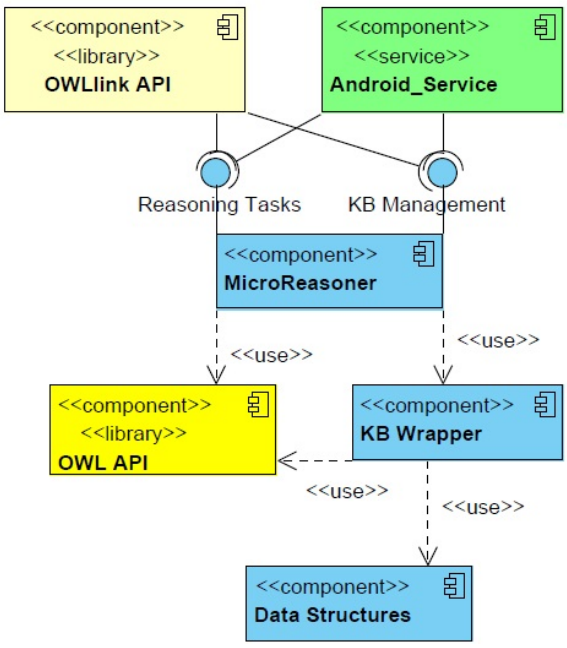
- **Concept Covering**: it allows to: cover (i.e., satisfy) features expressed in a request as much as possible, through the conjunction of one or more instances of a KB (seen as elementary building blocks) and provide explanation of the uncovered part of the request itself.

Reasoning services over ontologies are supported:

- **Ontology Coherence**: it is a simplified check with regard to Satisfiability, because it does not process ABox individuals. Since CNF normalization allows to identify unsatisfiable concepts, it is sufficient to normalize every concept during ontology parsing to locate unsatisfiabilities in the ontology.
- **Classification**: ontology classification computes the overall concept taxonomy induced by the subsumption relation, from $\top$ to $\bot$ concept.

2.3 Architecture

Mini-ME architecture is sketched as UML diagram:



- **Android Service**: implements a service any Android application can invoke to use the engine;
- **OWL API**: provides support for parsing and manipulating the OWL 2 language expressions;
- **OWLlink API**: provides support for the core OWLlink protocol, thus allowing requests for standard inferences only;
- **MicroReasoner**: reasoner implementation, exposing fundamental KB operations (load, parse), as well as inference tasks;
- **KB Wrapper**: implements KB management functions (creation of internal data structures, normalization, unfolding) and inference procedures on ontologies (classification and coherence check);
- **Data Structures**: in-memory data structures for concept manipulation and reasoning; at this level inference procedures on concept expressions (concept satisfiability, subsumption, abduction, contraction, covering) are implemented.

2.4 How Use

Mini-ME was developed in Java using Android SDK Tools, Revision 12, corresponding to Android Platform version 2.1 (API level 7), therefore it is compatible with all devices running Android 2.1 or later.
Mini-ME can be used:

1. as a library by calling public methods of the MicroReasoner component directly.
2. via OWLlink;
3. through the Android Service by Android applications;

2.4.1 With Protégé

Mini-ME has been integrated within the Protégé ontology editor [19] through the implementation of an OWL reasoner plugin, in order to facilitate ontology engineering and allow to design and prototype ubiquitous knowledge-based applications.

Protégé 4.3 on Windows 8.1

- Download **Protégé 4.3** [20] without `JVM`.

- Open `cmd`, navigate to the folder where the installation file was downloaded and execute:
`install_protege_4.3.exe LAX_VM "JAVA_HOME\java.exe"`
`JAVA_HOME` is the path where the java executable is located, for example: `"C:\Program Files\Java\jre1.8.0_111\bin"`

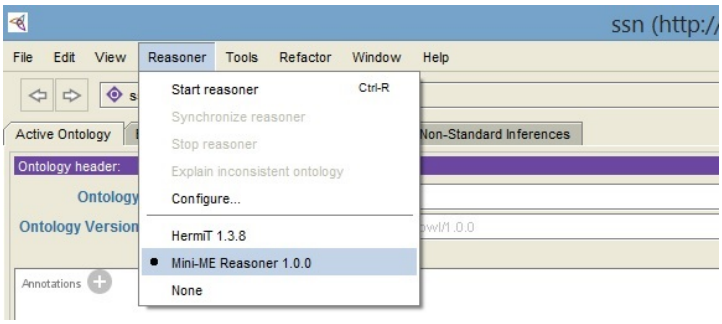
- Follow the instructions and select the path of the current installed version of Java, like above.

- If you have installed `JAVA 8` you must change the `felix.jar` file, in the folder `bin` where you have installed Protege 4.3 (for example `"C:\Program Files\Protege_4.3\bin"`), by a new one copied from a Protége 5.0 directory (download here the last version for example 5.6.2 [14]).
Rename `org.apache.felix.main.jar` in `felix.jar` and replace this with the previous version in the folder `bin`.

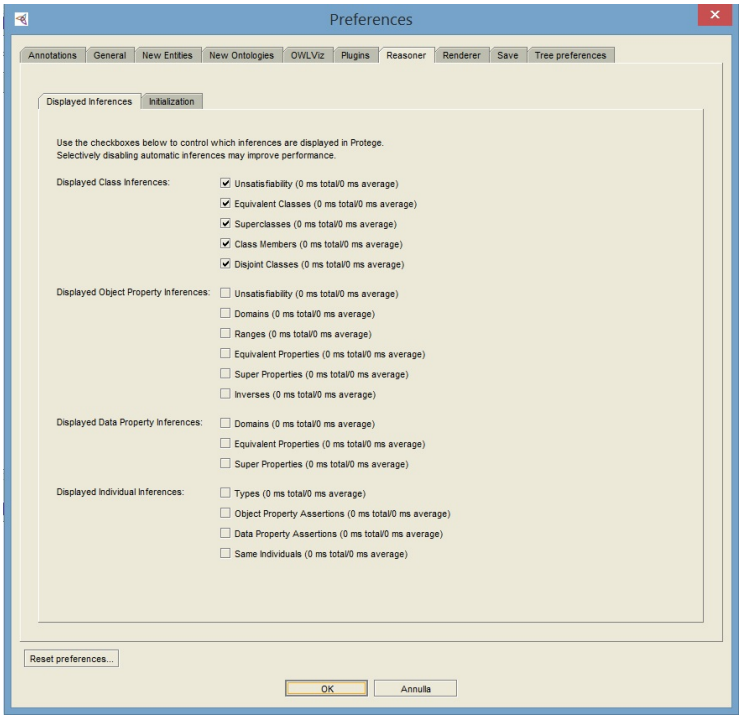
- To use Mini-ME with Protégé, simply download the plugins
(**it.poliba.sisinflab.minime.protege.reasoner.jar** [10] and
it.poliba.sisinflab.minime.protege.plugin.jar [9]) and place the files in the plugins directory of your Protégé installation (for example `"C:\Program Files\Protege_4.3\plugins"`).

- Now execute the `Protege.exe`.

- Now you can select the **Mini_ME Reasoner 1.0.0** in the Reasoning menu:

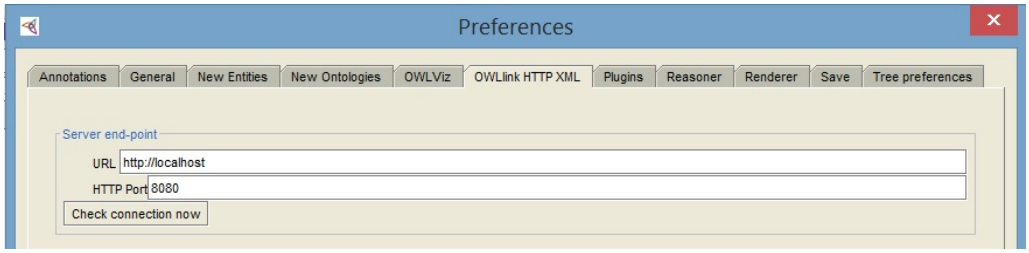


- In order to minimize the number of error messages, it is recommended to switch off some of the inference types. Go to `Reasoner > Configure > (Reasoner Tab) Displayed Inferences` and uncheck: all `Displayed Object Property Inferences`, all `Displayed Data Property Inferences` and all `Displayed Individual Inferences`.

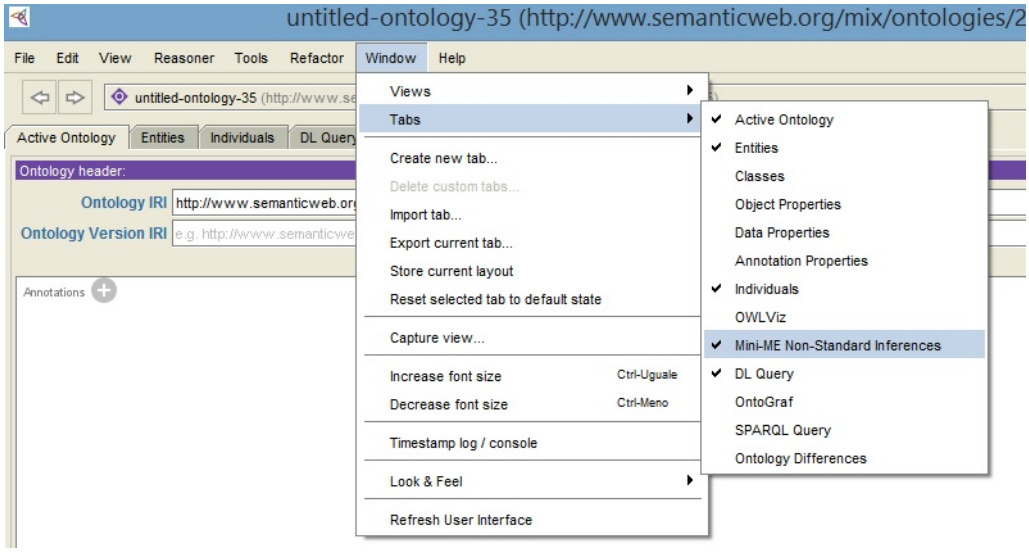


- Support for the `OWLlink protocol` was also integrated, based on the

OWLlink API (if you don't have you can download here [18] and place the files in the plugins directory of your Protégé installation). The Protégé OWLlink configuration panel (go to `Reasoner > Configure > OWLlink HTTP XML`), can be used to set up an OWLlink connection with the Mini-ME Reasoner.



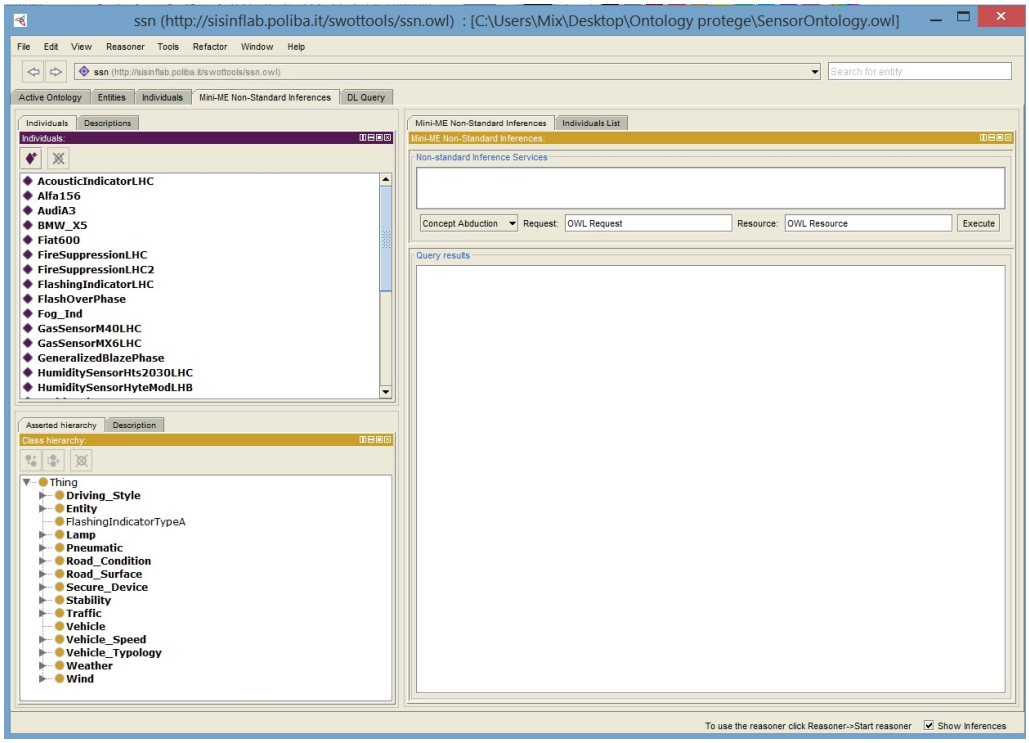
- The Mini-ME Non-Standard Inferences plugin will appear as an option the `Window > Tabs` menu.



- Open and load an Ontology (for example `SensorOntology.owl` [23]).

- Start the reasoner after you are sure that you have selected the Mini-ME Reasoner 1.0.0 (`Reasoner > Start reasoner`).

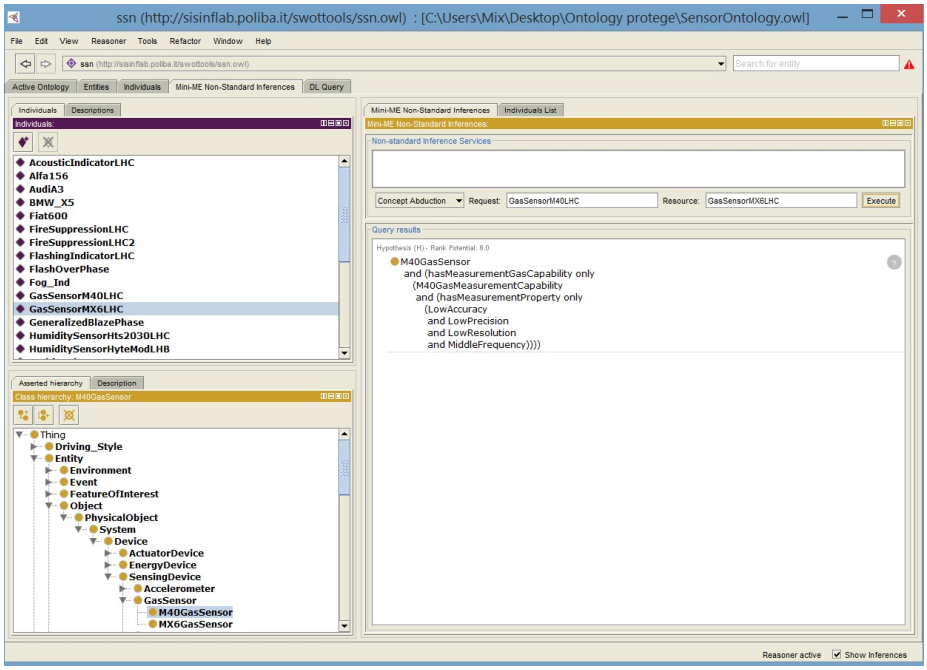
- A Protégé Tab Widget plugin has been developed to allow non-standard inferences through a user-friendly GUI.



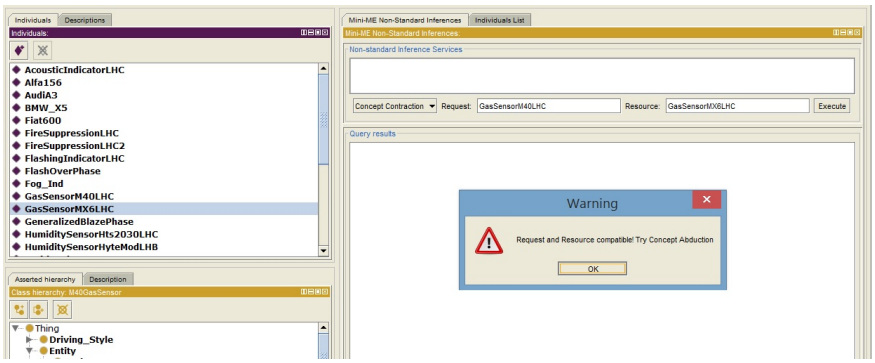
It consists of:

- **OWLIndividualsList** and **OWLIndividualsTypes** tabs, showing all individuals within the KB and their descriptions;
- **OWLAssertedClassHierarchy** and **OWLClassDescription** tabs, containing the class taxonomy and the description of selected classes;
- an **input box** used to select the inference service to execute, the request D and (in case of Concept Abduction and Concept Contraction) the supply S. Both can be selected from the OWLIndividualsList through drag-and-drop. For Concept Covering it is instead possible to select a subset of individuals through the **Individuals List panel**;
- a **results area** showing the output of the selected inference service.

- For example if you want have the result of **abductive inference** you should select an individual (i.e. `GasSensorM40LHC`) as **Request** and select an individual as supplied **Resource** (i.e. `GasSensorMX6LHC`) and **Execute**.

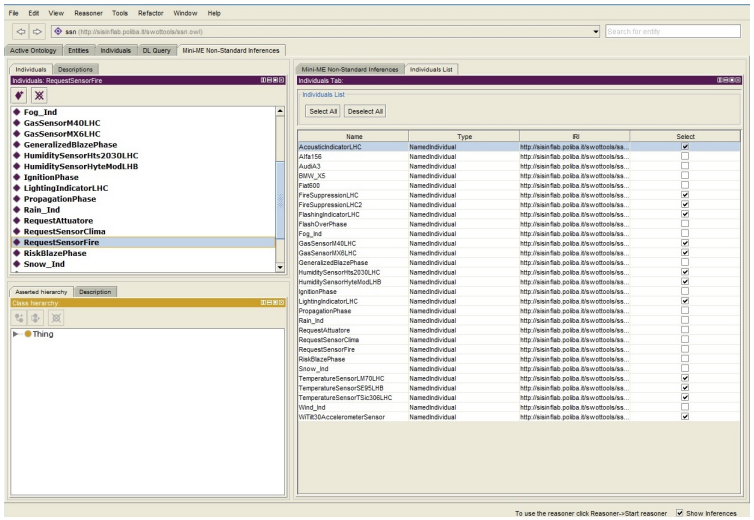


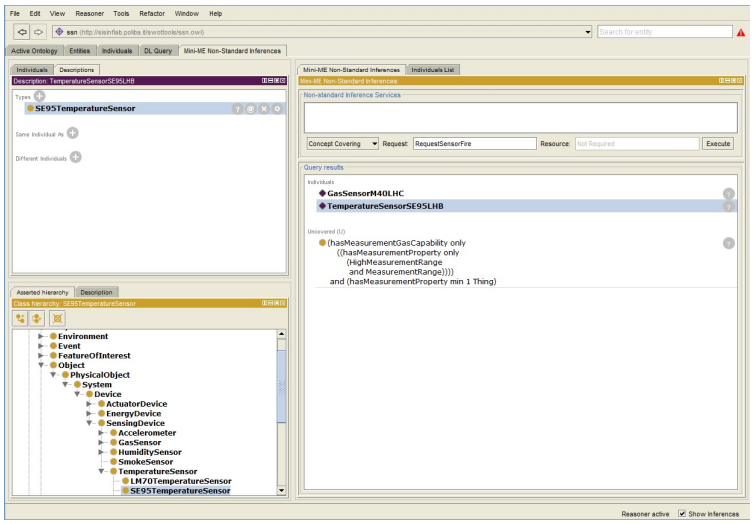
- Of course, since the two individuals are compatible, you can't do the **Contraction**.



- If you want have the result of **covering inference**, you should select an individual as **Request** (i.e. `RequestSensorFire`) and in this case the

supplied **Resource** is not required, but you can select a subset of individuals from the **Individuals List**, used as available resources of concept covering.





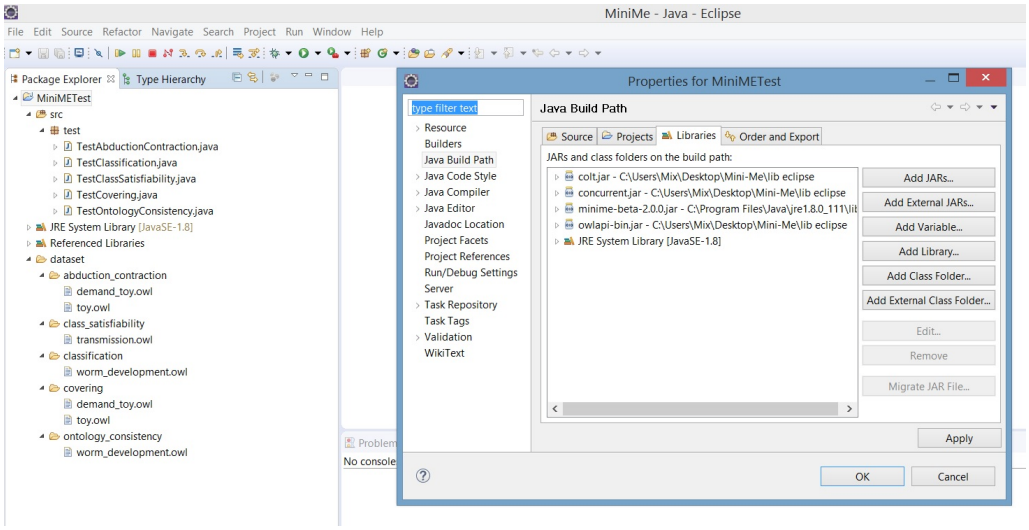
2.4.2 As library with Eclipse

Eclipse neon 3 on Windows 8.1

- Download **Eclipse neon 3** [5].
- Create a new Java Project and create a new package called `test`.
- Download the sample files here [24] and paste all of them in the package `test`.
- Download the folder with the Ontologies [3] and paste the folder `dataset` in the project.
- Download all necessary libraries:

  - The reasoner **Minime-beta-2.0.0** (**minime-beta-2.0.0.jar** [13]).
  - **Colt 1.2.0 API** (**colt.jar** and **concurrent.jar** in `lib` folder [4]).
  - **OWL API 3.2.4** (**owlapi-bin.jar** [16]).

- Import all these libraries: click right button on the project `> Proprieties > Java Build Path > Add External JARs` and select the four jar files downloaded earlier.



- Whereas we have already seen the example of Abduction, Contraction and Covering with Protégé, now we focus in the other concepts. For example **Ontology Consistency**:

```
1  public class TestOntologyConsistency {
2
3    static String ONTO_DIR = "./dataset/
        ontology_consistency/worm_development.owl";
4    static OWLOntology onto = null;
5    static OWLReasoner reasoner = null;
6
7    public static void main(String[] args){
8
9      try{
10       OWLOntologyManager manager = OWLManager.
           createOWLOntologyManager();
11       File onto_path = new File(ONTO_DIR);
12       //Load ontology from an input file
13       OWLOntology onto = manager.
           loadOntologyFromOntologyDocument(onto_path);
14       //Create the factory
15       MicroReasonerFactory reasonerFactory = new
           MicroReasonerFactory();
16       //Return an instance of OWLReasoner class that
           represents our Mini-ME reasoner
17       reasoner = reasonerFactory.createMicroReasoner(
           onto);
18       //The returned value is a boolean
19       boolean result = reasoner.isConsistent();
20       if(result){
21         System.out.println("The TBox of the ontology
             worm_development is consistent.");
22       }else {
23         System.out.println("The TBox of the ontology
             worm_development is not consistent.");
24       }
25
26     }catch(OWLOntologyCreationException e){
27         e.printStackTrace();
28     }
```

```
29    }
30  }
```

In this case we have in output:
`The TBox of the ontology worm_development is consistent.`

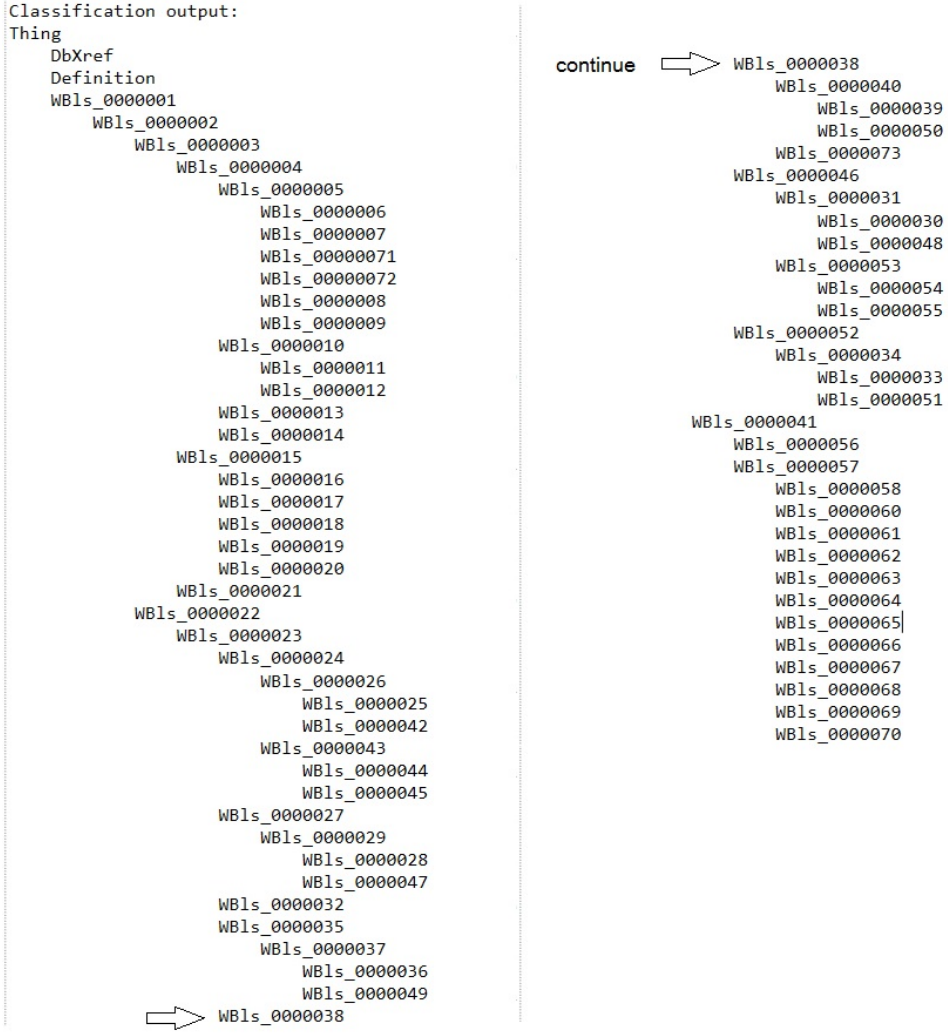
- Another example is the concept of **Classification**:

```
1  public class TestClassification {
2
3    static String ONTO_DIR = "./dataset/classification/
       worm_development.owl";
4    static OWLOntology onto = null;
5    static OWLReasoner reasoner = null;
6
7    public static void main(String[] args){
8
9      try{
10       OWLOntologyManager manager = OWLManager.
           createOWLOntologyManager();
11       OWLDataFactory owlDataFactory = OWLManager.
           getOWLDataFactory();
12       File onto_path = new File(ONTO_DIR);
13       //Load ontology from an input file
14       OWLOntology onto = manager.
           loadOntologyFromOntologyDocument(onto_path);
15       //Create the factory
16       MicroReasonerFactory reasonerFactory = new
           MicroReasonerFactory();
17       //Return an instance of OWLReasoner class that
           represents our Mini-ME reasoner
18       reasoner = reasonerFactory.createMicroReasoner(
           onto);
19       //Print classification result on the standard
           output.
20       System.out.println("Classification output:");
21       OWLClass top = owlDataFactory.getOWLThing();
22       try {
23         printClass(top, 0);
24       } catch (IOException e) {
```

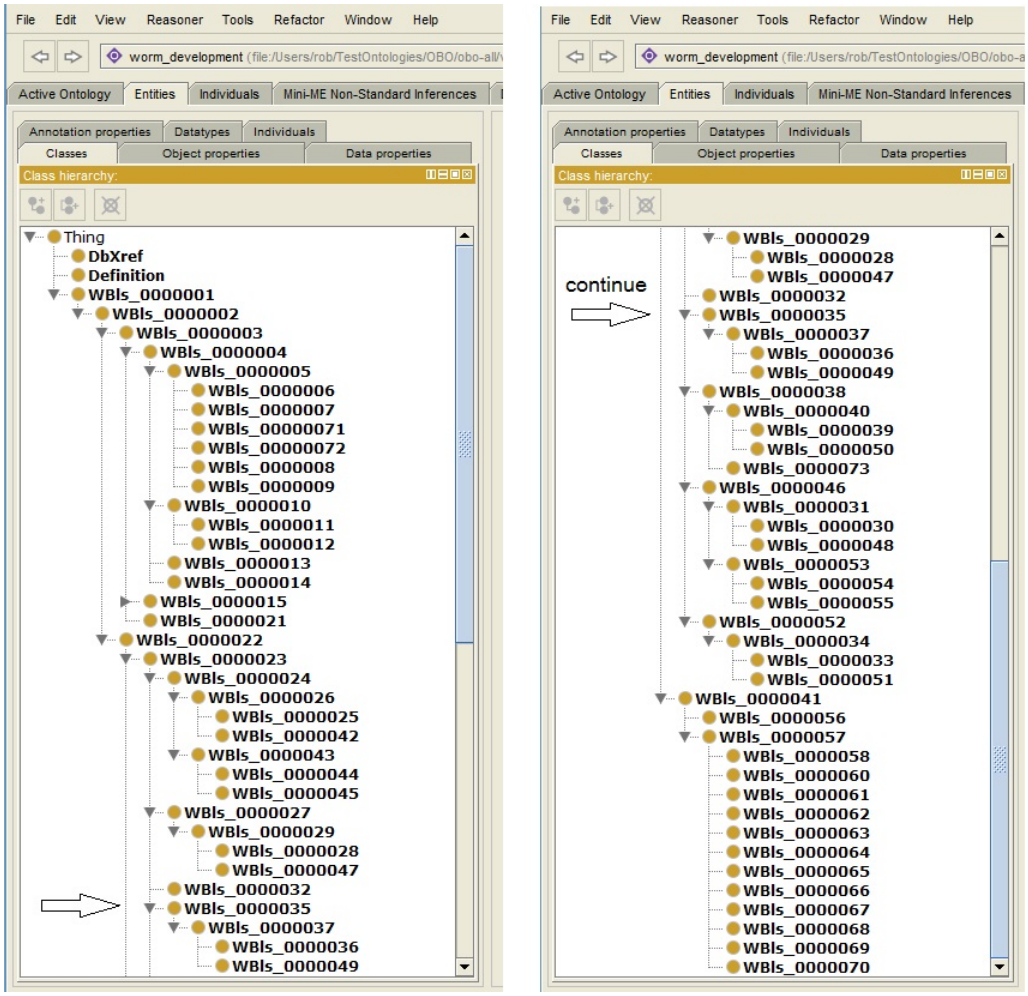
```
25          e.printStackTrace();
26        }
27
28      }catch(OWLOntologyCreationException e){
29        e.printStackTrace();
30      }
31    }
32
33    private static void printClass(OWLClass top, int
         depth) throws IOException{
34      if (!top.getIRI().getFragment().equals("Nothing")){
35        String line = "";
36        for(int i=0; i<depth; i++){
37          line += "    ";
38        }
39
40        line += top.getIRI().getFragment();
41        System.out.println(line);
42
43        ArrayList<String> classes = new ArrayList<String
             >();
44        HashMap<String, OWLClass> name = new HashMap<
             String, OWLClass>();
45        Iterator<Node<OWLClass>> it = reasoner.
             getSubClasses(top, true).iterator();
46        while(it.hasNext()){
47          OWLClass cls = it.next().
               getRepresentativeElement();
48          String className = cls.getIRI().getFragment();
49          if (className != null){
50            classes.add(className);
51            name.put(className, cls);
52          }
53        }
54        Collections.sort(classes);
55
56        for(int i=0; i<classes.size(); i++){
57          OWLClass subs = name.get(classes.get(i));
58          printClass(subs, depth+1);
```

```
59          }
60       }
61    }
62
63 }
```

In this case we have in output:



We would have had the same result thanks the graphic `Class hierarchy` in Protégé:



- You can see the concept of **Abduction**, **Contraction**, **Class Satisfiability** and **Covering** in the other examples.

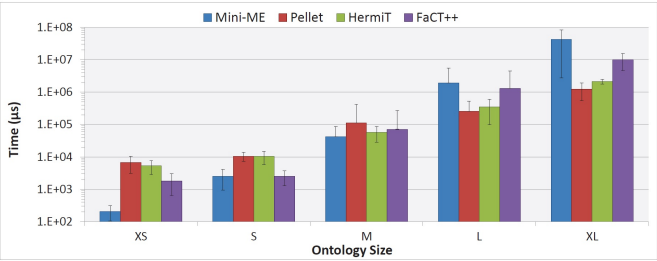
- You can do many other tests with innumerable ontologies available online, in particular further tests have been made with ontologies that can be found at this address:[2].

2.5 Compared with FaCT++, HermiT and Pellet on PC

Mini-ME was compared on PC with **FaCT++** [6], **HermiT** [7] and **Pellet** [17]. All reasoners were used via the OWL API on a PC testbed (Intel Core i7 CPU 860 at 2.80 GHz (4 cores/8 threads) with 8 GB DDR3-SDRAM (1333MHz) memory, 1TB SATA (7200 RPM) hard disk, 64-bit Microsoft-Windows 7 Professional and 32-bit Java 7 SE Runtime Environment,build 1.7.0_03-b05).
The reference dataset was taken from [2]: it is composed of 214 OWL ontologies with different size, expressiveness and syntax.
For each reasoning task, two tests were performed: (1) correctness of results and turnaround time; (2) memory usage peak.
For turnaround time, each test was repeated four times and the average of the last three runs was taken.
For memory tests, the final result was the average of three runs.
The ontologies have been divided into five categories, based on the number of concepts:

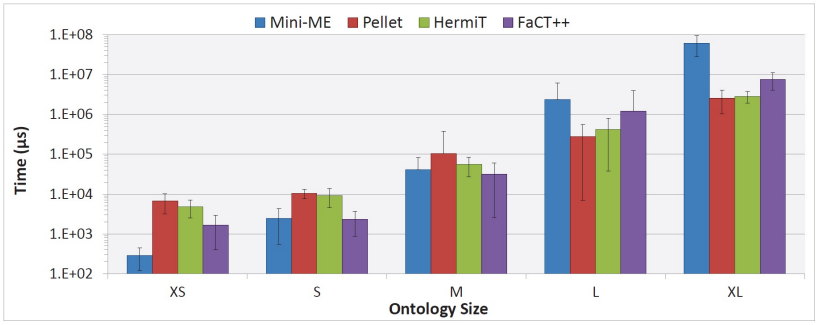
- **Extra Small** (XS): fewer than 10 concepts;
- **Small** (S): between 10 and 100 concepts;
- **Medium** (M): between 100 and 1000 concepts;
- **Large** (L): between 1000 and 10000 concepts;
- **Extra Large** (XL): more than 10000 concepts;

For **Classification**, performance was measured only for the 73 ontologies correctly classified by all reasoners.

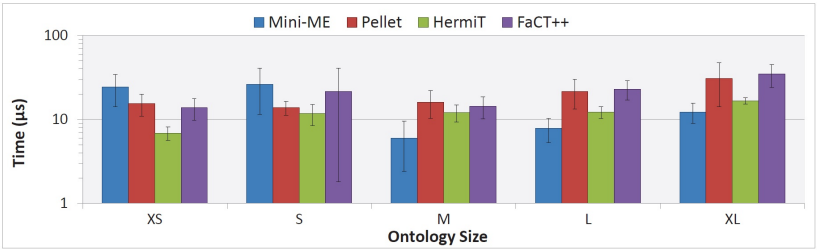


Mini-ME is very competitive for small-medium ontologies but less for large ones. This can be considered as an effect of the Mini-ME design, which is optimized to manage elementary TBoxes. The others exhibited a similar trend, with Pellet faster than the other engines for large ontologies.

For **Ontology Satisfiability** all reasoners presented performance similar to the ones for classification. In fact ontology satisfiability test implies loading, classifying and checking consistency of all concepts in the ontology with the first two steps requiring the larger part of the time.

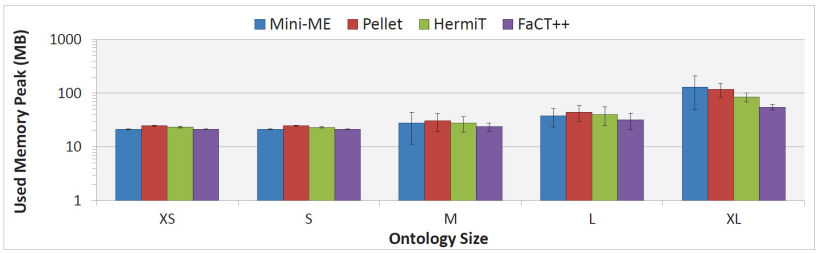


For **Class Satisfiability**, performances are basically similar, with times differing only for few microseconds and no reasoner consistently faster or slower. Moreover, the chart suggests no correlation between the time and the ontology size, whereas it is in direct ratio to the complexity of the class description and to its depth in the taxonomy.



The classification was verified as the most memory-intensive task, so the **memory usage** calculations were made during the classification. For small-medium ontologies, used memory is roughly similar for all reasoners. Mini-ME provides good results, with slightly lower memory usage than Pellet and HermiT and on par with FaCT++. Also for large ontologies Mini-ME

results are comparable with the other reasoners, but in this case FaCT++ has the best performance.



Conclusion

When processing semantic-based information to infer entailed implicit knowledge, painstaking optimization is needed to achieve acceptable performance for adequately expressive languages. This is specifically true in case of logic-based matchmaking for mobile computing, which is characterized by resource limitations affecting not only processing, memory and storage capabilities, but also energy consumption.
Most mobile inference engines currently provide only rule processing for entailments materialization in a KB, therefore they only support standard inference services such as satisfiability and subsumption, which provide only binary "yes/no" answers. So resulting unsuitable, in fact, non-standard inferences like abduction and contraction are needed to support approximate matches, semantic ranking and explanations of outcomes.
By limiting expressiveness to the $\mathcal{AL}$ language, acyclic, structural algorithms were adopted reducing standard and non-standard inference tasks to set-based operations. KB management and reasoning were then executed through a data storage layer, based on a mobile RDBMS (Relational DBMS). Such an approach was further investigated, by increasing the expressiveness to $\mathcal{ALN}$ DL and allowing larger ontologies and more complex descriptions, through the adoption of bothmobile OODBMS (Object-Oriented DBMS) and performance optimized data structures.
Therefore the non-standard services of Mini-ME, compared to other reasoners, are more useful in ubiquitous scenarios, where mobile agents must provide quick decision support and/or on-the-fly organization in intrinsically unpredictable environments like the ubiquitous ones.
Several case studies and prototypes have been developed, including ubiquitous commerce, ambient intelligence and infomobility services and home and building automation.

Bibliografia

[1] *Colt API*. 2017. URL: https://dst.lbl.gov/ACSSoftware/colt/api/.

[2] *Dataset for compared with HermiT, Pellet and Fact++*. URL: http://www.cs.ox.ac.uk/isg/conferences/ORE2012/materials.html.

[3] *Dataset with ALN Ontologies*. URL: http://sisinflab.poliba.it/swottools/minime/download/dataset.zip.

[4] *Download Colt 1.2.0 API*. URL: http://dst.lbl.gov/ACSSoftware/colt/colt-download/releases/colt-1.2.0.zip.

[5] *Eclipse neon 3*. URL: http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/neon3.

[6] *Fact++ version 1.6.3 with OWL API 3.4*. URL: http://owl.man.ac.uk/factplusplus/.

[7] *HermiT version 1.3.8*. URL: http://hermit-reasoner.com/.

[8] *Internet of Things. Wikipedia*. URL: https://en.wikipedia.org/wiki/Internet_of_things.

[9] *it.poliba.sisinflab.minime.protege.plugin.jar*. URL: http://sisinflab.poliba.it/swottools/minime/download/protege/it.poliba.sisinflab.minime.protege.plugin.jar.

[10] *it.poliba.sisinflab.minime.protege.reasoner.jar*. URL: http://sisinflab.poliba.it/swottools/minime/download/protege/it.poliba.sisinflab.minime.protege.reasoner.jar.

[11] Thorsten Liebig et al. «OWLlink». In: *Semantic Web – Interoperability, Usability, Applicability* 2.1 (2011), pp. 23–32.

[12] *Mini-ME*. 2017. URL: http://sisinflab.poliba.it/swottools/minime/.

[13] *Mini-Me beta 2.0.0*. URL: http://sisinflab.poliba.it/swottools/minime/download/minime/minime-beta-2.0.0.jar.

[14] *org.apache.felix.main.jar*. 2017. URL: https://mvnrepository.com/artifact/org.apache.felix/org.apache.felix.main.

[15] *OWL API*. URL: https://github.com/owlcs/owlapi/wiki.

[16] *OWL API 3.2.4*. URL: https://sourceforge.net/projects/owlapi/files/OWL%20API%20%28for%20OWL%202.0%29/3.2.4/owlapi-3.2.4.zip/download.

[17] *Pellet version 2.3.1*. URL: http://clarkparsia.com/pellet/.

[18] *Protege plugin and the binary OWLlink API as needed for the plugin*. URL: https://sourceforge.net/projects/owllink-owlapi/files/OWLlink%20for%20Protege%204.1/0.7.1/org.semanticweb.owllink.protege.jar/download.

[19] *PROTÉGÉ*. 2017. URL: http://protege.stanford.edu/.

[20] *PROTÉGÉ 4.3*. 2017. URL: http://protege.stanford.edu/download/protege/4.3/installanywhere/Web_Installers/.

[21] *Semantic Web*. URL: https://www.w3.org/standards/semanticweb/.

[22] *Semantic Web of Things. SisInfLab*. URL: http://sisinflab.poliba.it/swottools/.

[23] *SensorOntology.owl*. URL: http://sisinflab.poliba.it/swottools/minime/download/ontology/SensorOntology.owl.

[24] *Test Mini-ME 2.0.0 source code*. URL: http://sisinflab.poliba.it/swottools/minime/download/examples-minime-beta2.zip.