# MoK matching function: semantics on resource-constrained devices

Patryk Wojtowicz, Silvia Vandi, Davide Leardini

# 1 Introduction

Reasoners should play a vital role in developing and using an ontology written in OWL [18] (Web Ontology Language) a family of knowledge representation languages for authoring ontologies. Automated reasoners such as Pellet, FaCT++, HermiT, ELK and so on take a collection of axioms written in OWL and offer a set of operations on the ontologys axioms. The most noticeable of which from a developers perspective is the inference of a subsumption hierarchy for the classes described in the ontology.

## 1.1 Our work

We started our work with the analysis of a list of 57 reasoners. We have looked each of them to see if they are open source, Android compatible, ~~wrote~~ in Java and if yes, which API are usable. This research work has been divided equally between the ~~nunber of reasoners~~. After this work we have noticed that the most efficient, effective, famous, supported and ~~usefull~~ reasoners are JFact, Hermit and Pellet. Then each of us chose a reasoner and started some intense research on it. The paragraphs below are the results of these research, plus a section containing our tests of these three reasoners on a windows pc and on some Android devices.

## 1.2 Naming convention

There are different types of Description Logics [6] and there is an informal naming convention commonly used which roughly describes the operators allowed.
The expressivity is encoded in the label for a logic starting with one of the following basic logics:

- $\mathcal{AL}$ - Attributive language. This is the base language which allows atomic negation (negation of concept names that do not appear on the left hand side of axioms, concept intersection, universal restrictions and limited existential quantification.

- $\mathcal{FL}$ - Frame based description language. This allows concept intersection, universal restrictions, limited existential quantification and role restriction.
- $\mathcal{EL}$ - Allows concept intersection and existential restrictions (of full existential quantification).

All of these are followed by any of the following extensions:

- $\mathcal{F}$ - functional properties, a special case of a unique quantification;
- $\mathcal{E}$ - full existential qualification;
- $\mathcal{U}$ - concept union;
- $\mathcal{C}$ - complex concept negation;
- $\mathcal{H}$ - role hierarchy;
- $\mathcal{R}$ - limited complex role includes axioms (reflexivity and irreflexivity or role disjointness);
- $\mathcal{O}$ - nominals (enumerated classes of object value restrictions);
- $\mathcal{I}$ - inverse properties;
- $\mathcal{N}$ - cardinality restrictions;
- $\mathcal{Q}$ - qualified cardinality restrictions (available in OWL 2);
- $(\mathcal{D})$ - use of datatype properties, data values or data types.

In this classification there are also some exception, but for that we refere to the documentation.

# 2 Reasoning

## 2.1 Reasoning with JFact

JFact is a port of FaCT++ to Java. FaCT++, successor of Fact, is a DL reasoner designed as a platform for experimenting with new tableaux algorithms and optimisation techniques. It incorporates most of the standard optimisation techniques, including those introduced in the FaCT system, but also employs many new ones. This includes a new ”ToDo list” architecture that is better suited to more complex tableaux algorithms (such as those used to reason with OWL ontologies), and allows for a wider range of heuristic optimisations. FaCT++ reasoner is a free reasoner implemented in C++.

JFact design matches FaCT++ design very closely, except for the low level details where Java offers different ways of doing things. There are a lots of JFact classes and methods whitch are the same, or similar to, ~~respect~~ FacT++ one. Both also share the same license, therefore they are available under the Lesser GPL license.

One difference is that JFact does not include some parts of FaCT++: the DIG interface [17], which defines a simple protocol (based on HTTP PUT/GET) along with an XML Schema that describes a concept language, and the command line supports are absent.

JFact supports full OWL 2 with reasoning based on a tableaux algorithm, and the supported expressivity for reasoning is $\mathcal{SROIQ}(\mathcal{D}$n).

On GitHub we can find a JFact repository [8].

## 2.2 Reasoning with HermiT

HermiT is an open-source reasoner for ontologies written using the Web Ontology Language (OWL). Given an OWL file, HermiT can determine whether or not the ontology is consistent or identifies subsumption relationships between classes [9].

This is the first publicly-available OWL reasoner based on a new ”hypertableau” calculus which

provides much more efficient reasoning than any previously-known algorithm because it greatly reduces the number of possible models which must be considered during the test(down to only a single possibility for a significant subset of ontologies). Ontologies which previously required minutes or hours to classify can often be classified in seconds by HermiT.

~~The sources of complexity in state of the art OWL reasoners is particularly important in practice, and it is achieved in HermiT with an improved blocking strategy,~~ named "anywhere blocking", and an optimization that limits the size~~s~~ of models which are constructed and tries to reuse existing individuals rather than generating new ones.

This reasoner is the first one able to classify a number of ontologies which had previously proven too complex for any available system to handle and also incorporates a number of other new optimizations, such as a more efficient approach to handling nominals, and various techniques for optimizing ontology classification.

Finally, HermiT makes use of a new and highly-efficient approach to handling nominals in the presence of number restrictions and inverse roles~~; T~~he authors of this research expected that this will allow ontology authors to make much freer use of nominals than has been possible to date. This combination of fundamental algorithmic improvements also enables a range of additional optimizations.

~~The~~ HermiT supported expressivity for reasoning is $\mathcal{SHOIQ}+$. It uses direct semantics, which defines the formal, model-theoretic semantics of OWL 2 [10], and passes all OWL 2 conformance tests for direct semantics reasoners. ~~Its~~ released HermiT 1.3.8 under the GNU Lesser General Public License (LGPL). The release should be compatible with Java 1.5 or higher. HermiT 1.3.8 uses the OWL API 3.4.3, which is backwards compatible with the OWL API 3.3.x, 3.2.x and 3.1.x, but not backwards compatible with the OWL API 3.0.x. Since version 1.1, HermiT can handle DL Safe rules and the rules can directly be added to the input ontology in functional style or other OWL syntaxes supported by the OWL API (see A Syntax for Rules in

OWL 2).

## 2.3 Reasoning with Pellet

Pellet is a complete OWL-DL reasoner with extensive support for reasoning with individuals (including nominal support and conjunctive query), user-defined datatypes, and debugging support for ontologies [11]. It offers a panoply of features including conjunctive query answering, rule support, E-Connection reasoning, and axiom pinpointing, among others. To make its reasoning capabilities easily accessible to users, Pellet provides various interfaces including a command-line interface, an interactive Web form for zero-install use, DIG server [17] implementation, and API bindings for RDF/OWL toolkits Jena and Manchester OWL-API [12].

Pellet is written in Java, requiring a Java 5 (or greater) JVM, and it is open source under Dual-license, Proprietary license available for commercial applications and AGPL license for open-source applications.

It implements several extensions to OWL-DL including a ~~combination formalism~~ for OWL-DL ontologies, a non-monotonic operator, and preliminary support for OWL/Rule hybrid reasoning. It supports ~~all of~~ OWL DL ($\mathcal{SHOIN}(\mathcal{D})$) and has been extended to OWL 2 ($\mathcal{SROIQ}(\mathcal{D})$). Pellet so supports OWL 2 profiles including OWL 2 EL. OWL-DL restricts OWL-Full ontologies in several different ways, and Pellet relaxes some of the OWL-DL restrictions [13]:

- pellet uses punning semantics of OWL 1.1 to partially overcome the vocabulary separation restriction;
- pellet fully supports inverse functional datatype properties;
- annotations have no effect on reasoning results and will be completely ignored by Pellet regardless of the configuration option;

- pellet extends punning to built-in RDFS vocabulary (for example: if `rdf:List` is used in individual assertions it will be treated as a new class);

- all axioms need to be well-formed and any other axiom will be ignored. However, some URIs might be untyped and Pellet will ~~uses~~ some heuristics to infer the type for those URIs. For example, if a URI is used in a `owl:subClassOf` then that URI will be treated as if it is defined as an OWL class.

Pellet provides the ”standard” set of Description Logic inference services:

- consistency checking , which ensures that an ontology does not contain any contradictory facts;

- concept satisfiability , which checks if it is possible for a class to have any instances. If class is unsatisfiable, then defining an instance of the class will cause the whole ontology to be inconsistent.

- classification, which computes the subclass relations between every named class to create the complete class hierarchy. The class hierarchy can be used to answer queries such as getting all or only the direct subclasses of a class.

- realization, which finds the most specific classes that an individual belongs to; or in other words, computes the direct types for each of the individuals. Realization can only be performed after classification since direct types are defined with respect to a class hierarchy. Using the classification hierarchy, it is also possible to get all the types for that individual.

Pellet functionality can be accessed via Java API or via other three different APIs:

- Internal Pellet API - API used by the reasoner, designed for efficiency but not for usability. It uses ATerm library for representing terms, and it allows Fine-grained control over reasoning. By missing various features (parsing and serialization), lack in usability;

- Manchester OWL API - API closely tied to OWL structural specification. It supports many syntaxes (RDF/XML, OWL/XML,OWL functional, Turtle, ...), it has Native SWRL support and it allows Integration with reasoners. Despite having the feature of being a OWL-centric API, it lacks in stability and has no SPARQL support;

- Jena API - An RDF API with OWL extensions, with a In-memory and persistent storage, Built-in rule reasoners and integrated with Pellet, with a SPARQL query engine. Despite being mature, stable and ubiquitous, it is not great for handling OWL.

On GitHub we can find a Pellet repository [14].

## 2.4 Running Reasoners on Android

Most of current popular semantic reasoners are implemented using Java and are usually used along with semantic APIs like OWL API or Jena. Android, which is a Linux-based operating system whose middleware, libraries, and APIs are written in C, supports Java code as it uses a Java-like virtual machine called Dalvik [2]. It runs ”dex-code” (Dalvik Executable), and Java bytecodes can be converted to Dalvik-compatible .dex files to be executed on Android. However, Dalvik does not align to Java SE and so it does not support Java ME classes, AWT, or Swing. Thus, although most of the semantic APIs and DL reasoners are implemented in Java, porting them into Android requires some human intervention.

JFact 0.9.15, which does not import any external libraries (except for the OWLAPI), works fine on Android because its code can be converted directly to Dalvik. In this way, the only things to do ~~is develop~~ an Android app that uses this reasoner by simply importing the JFact 0.9.1 and OWLAPI 3.4.3 .jar files.

On the other hands HermiT and Pellet cannot be converted directly to Dalvik. The problems that the authors had verify with Hermit are:

- references to unsupported Java classes;
- runtime error of Dalvik when unmarshalling the objects serialized by `dk.brics.automaton`.

Solution found:

- remove the debug package of HermiT and its references;
- reimplement some methods of JAutomata class;
- reimplement the marshalling/unmarshalling methods.

For Pellet they have noticed that they cannot be converted directly to Dalvik as it uses three libraries that reference unsupported Java classes:

- Jena - can be replaced by Androjena, a port of Jena to the Android platform (the last released version is version 0.5, which contains all the original code of Jena 2.6.2.);
- OWL API 2.2.0 - can be removed;
- JAXB - can be fixed:
  - remove the JAXB .jar file;
  - add to our Android project the source code of the java.xml.bind package and the Xerces library;
  - due to the limit of 65536 methods references per .dex file, add only the 9 classes that Pellet needs.

# 3 Comparison

## 3.1 Testing on MacBook Pro

All of this three reasoners are complete for OWL 2 and highly optmized, but still too slow for some ontologies.

In fact, as we can see in benchmarks like this [5], there were tests made on a 2.2 GHz MacBook Pro, and for the three different reasoners there were very differents value in terms of time.

The tests were made ~~over only ontologies contained single file~~, and all the data was ~~parsing~~ before running benchmarks. This parsing was made because most of the data used external ”imports”, so this was localized by parsing the ontologies, parsing all imports and merging the main and imported ontologies together using OWL API 2.2.1.

For these benchmarks, for each reasoner, was allocated 1 gigabyte of RAM and given 30 min to classify each ontology.

The test was composed from different parts:

- parsing the ontology using OWL API;
- loading the ontology into the reasoner;
- classification of the ontology;
- writing the classified taxonomy to disk.

It was misured the time for part 2 and 3. On average Pellet is the faster, Hermit takes about double of Pellet’s time, instead Fact++ crashed for the most of the trials, or took more then 30 mins.

In terms of correctness the results found that Fact++ and Hermit are the best. There were some cases in wich it appeared that the version of Pellet tested missed some interfaces. These

cases were investigated manuallly, and it does appeared that Pellet's results were incorrect while Fact++ and Hermit produce the correct results.

## 3.2 Testing on a Linux System

Benchmarks were also made[4] on a Linux 2.6.16.1 System with Java 1.5.0 Update 6 where the maximum heap space was set to 800 MB, and the maximum time was set to 5 mins. The tests here were composed from these parts:

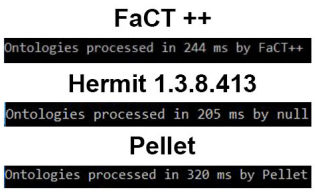
- creation of new istance;
- loading the ontology into the reasoner (No methods about optimization for the reasoners were called)
- classification of the ontology.

The test here were based on loading time and classification time.
We saw Pellet in comparison with other reasoners which aren't Hermit or Fact++. The reasoners tested here were Kaon2, Owlim, Pellet, RacerPro, Sesame. From these benchmarks it was deduced that Pellet was still on average while loading or with conjunctive query time. In some cases it reached the set out time, but still better then other reasoners.

## 3.3 Testing on Windows System

For testing the reasoners on Windows we ~~have~~ used Protege 5.0.0 beta [15], a desktop program developed by Stanford University, that was very useful to manage the ontologies. We have run the same Snap SPARQL query on the same ontology (http://owl.man.ac.uk/2006/07/sssw/people.owl) with three different reasoners (FACT++, Hermit 1.3.8.413 and Pellet) and we ~~have taken~~ note of the time each reasoner takes to evaluate them. For the test we ~~have~~ used an Asus PC I7

2.50 GHz with 8GB of RAM and Windows 10. First of all we ~~have taken note the~~ time every reasoner needs for loading the same ontology, and the results are shown in the picture below.




After we have run some simple Snap SPARQL query, below are shown both the queries and times taken from each reasoners.

Query number 1 returns a list of all the cats whoose names starts with ”T” and the respective owners.

```
SELECT ?personName ?petName ?gender
WHERE
{
   ?person rdf:type :person ;
         rdfs:label ?personName;
         :has_pet ?pet.
   ?pet rdf:type :cat ;
      rdfs:label ?petName
   FILTER (STRSTARTS(?petName, "T"))
}
ORDER BY ?petName
```

| | |
|---|---|
| FaCT++ | 21 ms |
| Hermit 1.3.8.413 | 11 ms |
| Pellet | 5ms |

Query number 2 retrieves every superclasses composing ~~of~~ ontology with relative URL.

```
SELECT ?x (STR(?lab) AS ?label) WHERE {
            ?x rdf:type owl:Class .
            OPTIONAL {?x rdfs:label ?lab}
}
ORDER BY ?label
```

| | |
|---|---|
| FaCT++ | 4 ms |
| Hermit 1.3.8.413 | 3 ms |
| Pellet | 1 ms |

Query number 3 returns all the common superclasses between magazine and tabloid.

```
SELECT ?superClass WHERE {
 ns:magazine  rdfs:subClassOf ?superClass .
 ns:tabloid rdfs:subClassOf ?superClass .
}
```

| | |
|---|---|
| FaCT++ | 0 ms |
| Hermit 1.3.8.413 | 1 ms |
| Pellet | 1 ms |

Query number 4 returns all the subclasses of person.

```
SELECT ?lcs WHERE {
   ?lcs  rdfs:subClassOf  ns:person
}
```

| | |
|---|---|
| FaCT++ | 19 ms |
| Hermit 1.3.8.413 | 2 ms |
| Pellet | 3 ms |

Unfortunately in this framework the command `FILTER` wasn't implemented, so we were unable to make more interesting queries, and we have decided to test the reasoners on Android. However from the tests using Protege we can assume that Hermit is the faster processing ontologies, but Pellet seems to be the faster evaluating queries.

### 3.3.1 Snap SPARQL

Snap-SPARQL is an open source Java framework for working with SPARQL and OWL which includes a parser, axiom template API, SPARQL algebra implementation, and graphical user interface components for reading, processing and executing SPARQL queries under the SPARQL 1.1 OWL Entailment Regime [16]. This framework is already included in Protege 5.0.0 beta, but on GitHub there are update files and binaries for the plug-ins [3]. Also on GitHub are available the API [1] for translating SPARQL queries into Snap-SPARQL ones.

## 3.4 Testing on Android System

For testing the reasoners on Android we ~~have~~ used an Eclipse plugin called Android Development Tools (ADT). First of all we ~~have~~ downloaded from GitHub the APIs of three different reasoners [8] [14] [7] and we ~~have done different applications for~~ JFact, Pellet and Hermit.
In ~~this~~ our simple applications we try to find the common superclass between two individuals of the same ontology which we have used for the tests on Windows. Since there isn't an automatic way to do it, we have to bypass the obstacle. We try to use `DefaultFixedManager()` to directly extract the statement, but it doesn't work, so to solve this problem we have to use a for cicle which iterates all the nodes and checks for the one we are looking for.

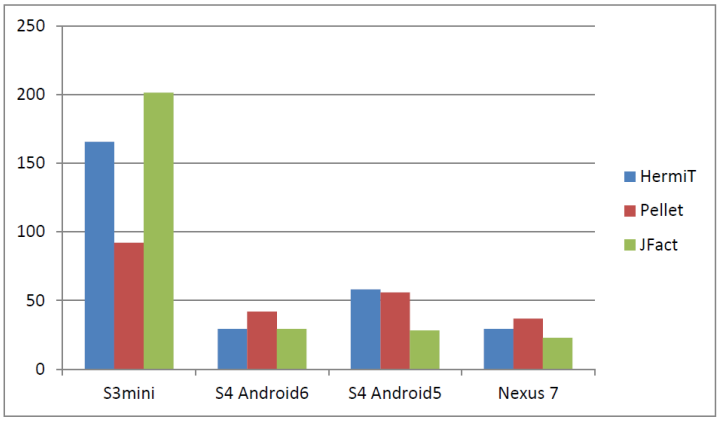
```
OWLNamedIndividual duckname = null;
for (OWLClass c : peopleOntology.getClassesInSignature()) {
    if (c.getIRI().getFragment().equals("duck")) {
        duckinstances = reasoner.getInstances(c, false);
        System.out.println(c.getIRI().getFragment());
        myText.append("\n" + c.getIRI().getFragment());
        for (OWLNamedIndividual i : duckinstances.getFlattened()) {
            System.out.println(i.getIRI().getFragment());
            myText.append("\n" + i.getIRI().getFragment());
            duckname = i;
        }
    }
}
myText.append("\n\n\n superclassi di: " + duckname + "\n\n\n" + reasoner.getTypes(duckname, false) + "\n\n");
```

With the method `getTypes()` we got an offset of classes which represents all the superclass starting from Thing and then we have to filter the subclasses which have node A as superclass, where node A represents the father node of the elements we are looking for.

We ~~have~~ tested these applications on a Samsung Galaxy S3 mini with Android 4.2.2, a Samsung Galaxy S4 with Android 6.0.1, a Samsung Galaxy S4 with Android 5.0.1 and a Google Nexus 7 1GEN with Android 5.1.1 . For each device we ~~have~~ started the same queries, ~~one which we want to have~~ the common superclass between "tabloid" and "dog", another one between "cow" and "duck" and the last one between "person" and "female".

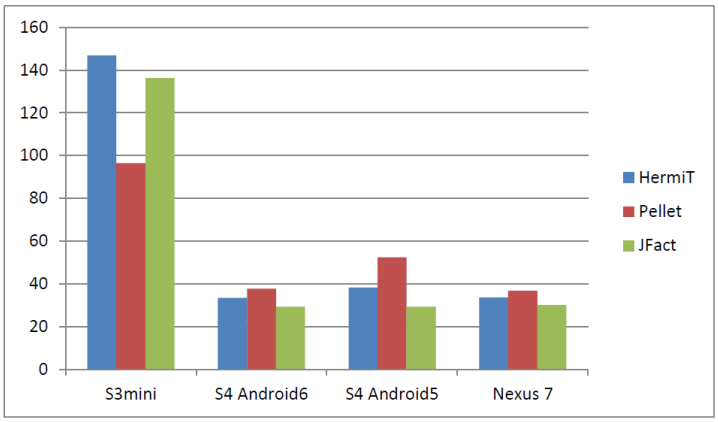
The results of the first query is `Thing`. Below are shown the table and histogram of times in milliseconds.

| Query 1 | S3mini | S4 Android6 | S4 Android5 | Nexus 7 |
|---|---|---|---|---|
| HermiT | 165,67 | 29,36 | 58,2 | 29,49 |
| Pellet | 92,27 | 41,94 | 56,09 | 36,83 |
| Jfact | 201,32 | 29,36 | 28,31 | 23,06 |




The results of the second query is `Animal`. Below are shown the table and histogram of times in milliseconds.

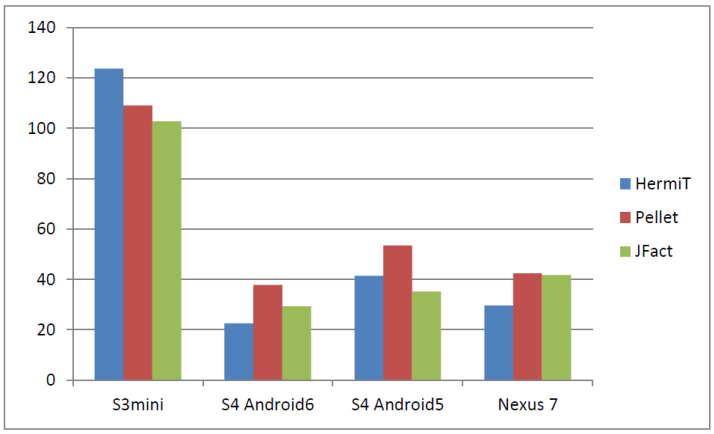
| Query 2 | S3mini | S4 Android6 | S4 Android5 | Nexus 7 |
|---|---|---|---|---|
| HermiT | 146,8 | 33,55 | 38,27 | 33,62 |
| Pellet | 96,46 | 37,74 | 52,42 | 36,77 |
| Jfact | 136,31 | 29,36 | 29,36 | 30,15 |




And the result of the third query is `grown up/pet owner`. Below are shown the table and

histogram of times in milliseconds.

| Query 3 | S3mini | S4 Android6 | S4 Android5 | Nexus 7 |
|---|---|---|---|---|
| HermiT | 123,73 | 22,55 | 41,41 | 29,62 |
| Pellet | 109,05 | 37,74 | 53,48 | 42,4 |
| Jfact | 102,76 | 29,36 | 35,13 | 41,71 |




From this test we can see that for devices like Samsung Galaxy S3 mini Pellet is the best, but for devices with better hardware component the ~~domination~~ of one reasoner compared to the others is not so clear.

### 3.4.1 How-to reproduce our tests on Android System

To repeat our tests it is necessary to have Eclipse with Android Development Tools plugin, which is directly downloadable from the program under the section Help - Install New Software. At the following link there is a step-by-step tutorial for this part.
(https://stuff.mit.edu/afs/sipb/project/android/docs/sdk/installing/installing-adt.html)
The API required for the reasoners are downloadable from GitHub (link in bibliografy [8] [14] [7]).
Finally our code is needed, which is downloadable here:
https://bitbucket.org/Bashnak/semantics-on-resource-constrained-devices/src
Once a new Java project is created, it is necessary to include all the libraries which can be found

in the package named libs. The nucleus of our test is all writen in the file named MainActivity.java, which is located in the package named main.

To modify the query's parameters it's needed to change the elements between quotes writen in lines which are specified in code with comments.

The screen that will be shown after running the application is as the example below.


The query result is shown at the bottom of the device or emulator screen.

# 4 Conclusion

In conclusion after all this research and tests, some made from us and others not, we can deduce that, at the state of the art, there isn't one more eligible or useful reasoner than the others. In some cases we have seen a decisive predominance of Pellet, but in most cases both Pellet and Hermit who Jfact are good choice.

# References


[1] "An API for parsing SPARQL queries". In: (2016). URL: https://github.com/protegeproject/snap-sparql-query.

[2] "Android goes Semantic: DL Reasoners on Smartphones". In: (2013). URL: http://www.cs.ox.ac.uk/isg/conferences/ORE2013/slides/ORE_Session4_Android_Reasoners.pdf.

[3] "Autoupdate files for Protege Desktop". In: (2016). URL: https://github.com/protegeproject/autoupdate.

[4] "Benchmarking OWL Reasoners". In: (2008). URL: http://ceur-ws.org/Vol-350/paper1.pdf.

[5] "Benchmarks". In: (2009). URL: http://www.hermit-reasoner.com/2009/JAIR_benchmarks/.

[6] "Description logic". In: (2016). URL: https://en.wikipedia.org/wiki/Description_logic.

[7] "Hermit repository". In: (2015). URL: https://github.com/robertoyus/HermiT-android.

[8] "JFact repository". In: (2016). URL: https://github.com/owlcs/jfact.

[9] "Knowledge Representation and Reasoning". In: (2012). URL: http://www.hermit-reasoner.com/.

[10] "OWL 2 Web Ontology Language: Direct Semantics". In: (2008). URL: https://www.w3.org/TR/2008/WD-owl2-semantics-20081008/.

[11] "Pellet". In: (2011). URL: https://www.w3.org/2001/sw/wiki/Pellet.

[12] "Pellet: A practical OWL-DL reasoner". In: (2007). URL: http://www.sciencedirect.com/science/article/pii/S1570826807000169.

[13] "Pellet: A Practical OWL-DL Reasoner". In: (2007). URL: http://www.cs.ox.ac.uk/people/bernardo.cuencagrau/publications/PelletDemo.pdf.

[14] "Pellet repository". In: (2015). URL: https://github.com/Complexible/pellet.

[15] "Protege". In: (2016). URL: http://protege.stanford.edu/products.php.

[16] "Snap-SPARQL: A Java Framework for working with SPARQL and OWL". In: (2015). URL: http://cgi.csc.liv.ac.uk/~valli/OWLED2015/OWLED_2015_paper_21.pdf.

[17] "The new DIG interface standard (DIG 2.0)". In: (2006). URL: http://dl.kr.org/dig/interface.html.

[18] “Web Ontology Language”. In: (2016). URL: https://en.wikipedia.org/wiki/Web_Ontology_Language.