# Semantic Reasoning on Mobile Devices: Do Androids Dream of Efficient Reasoners?

Carlos Bobed, Roberto Yus, Fernando Bobillo, Eduardo Mena

*Department of Computer Science & Systems Engineering, University of Zaragoza, Spain*

**Abstract**

The massive spread of mobile computing in our daily lives has attracted a huge community of mobile application (*apps*) developers. These developers can take advantage of the benefits of semantic technologies (such as knowledge sharing and reusing, and knowledge decoupling) to enhance their applications. Moreover, the use of semantic reasoners would enable them to create more intelligent applications capable of discovering new knowledge, inferred from the available information.

However, using semantic APIs and reasoners on current mobile devices is not a trivial task. In this paper, we show that the most popular current available Description Logics (DL) reasoners can be used on Android-based devices, and detail the efforts needed to port them to the Android platform. We also analyze the performance of these reasoners on current smartphones/tablets against more than 300 ontologies from the ORE 2013 ontology set, showing that, despite a notable difference with respect to desktop computers, their use is feasible.

*Keywords:* Reasoning, Semantic Web, Mobile Computing, Android

## 1. Introduction

In the last few years, we have witnessed a massive spread of mobile computing which is shaping our daily lives. This has been undoubtedly due to the ever increasing computing capabilities of mobile devices, and the almost pervasive connectivity that the current wireless networks provide us with. As a consequence, thousands of mobile applications (*apps*) have been developed. It seems a promising idea to enhance such applications by taking advantage of the well-known benefits of using ontologies in desktop computers [1], such as the improvement of knowledge sharing, reusing and maintenance, the decoupling of the knowledge from the application, or the possibility of discovering implicit knowledge by using semantic reasoners. However, while mobile applications have attracted a whole of attention, the use of semantic techniques in them is quite far from being usual.

The most direct way to add semantic reasoning to mobile applications would be, indeed, to rely on external servers which would perform all the calculations. However, in this ubiquitous and mobile scenario, where context-awareness and privacy preserving play a crucial role, sending sensitive data to a remote server might be an important privacy breach, and even sending non-sensitive data might be dangerous as it could enable the inference of sensitive information. Despite the fact that reasoning on mobile devices has been the subject of interest from the early stages of the Semantic Web [2, 3], it has not been until recently that mobile devices have been considered as effective reasoning platforms (in terms of CPU power, memory, and connectivity). Thus, current mobile devices make local reasoning feasible and open new opportunities in this area (e.g., mixing local with remote reasoning by determining what data can be disclosed and sent to the server and joining the results). Moreover, despite the communications advances, there are situations where connectivity with a server can be faulty or nonexistent, and therefore the only way to perform the reasoning is locally.

*Email addresses:* cbobed@unizar.es (Carlos Bobed), ryus@unizar.es (Roberto Yus), fbobillo@unizar.es (Fernando Bobillo), emena@unizar.es (Eduardo Mena)

As an example of mobile applications that are currently taking advantage of the use of local semantic reasoning, we can include Location-Based Services (LBS) providers, mobile health (m-health) systems, and privacy control applications:

- In LBS providers, it is often particularly important to make decisions considering the user context (e.g., deciding the most relevant information about means of transport at the moment). For instance, the *SHERLOCK* system [4] uses a semantic reasoner and background knowledge about means of transport to infer that both a cab and a tram are interesting for a certain mobile user, given the information obtained from sensors on his/her device such as the location and time. Other location-based semantic applications that could benefit from semantic reasoners are *DBpedia Mobile* [5] and *mSpace Mobile* [6].

- In the field of m-health, semantic reasoners have been proposed as part of monitoring systems [7, 8] and clinical decision support systems [9] due to the sensitivity of the information managed. *Rafiki* [9] is an example of the latter one that uses a semantic reasoner within the mobile device to infer possible diseases for a patient in a rural area, where connectivity is usually non-existent, given his/her symptoms and context.

- Regarding privacy control applications, local semantic reasoning could also help systems to preserve the privacy of users by inferring whether information about him/her should be shared with other people or applications according to the user context. This has been investigated for smartphones [10] and Google Glass [11].

We expect more applications such as those presented to appear as we have detected the interest in the use of semantic reasoners on mobile devices (for example, the webpage dedicated to our research on Semantic Web on mobile devices [12] had 1500 visits during 2014, and around 2800 during the first semester of 2015).

However, the development of semantic mobile applications has not (yet) spread due, in part, to the fact that there are currently no remarkable efforts to enable mobile devices with semantic reasoning capabilities. A first possibility is developing new reasoners specifically designed for mobile devices. Examples of this alternative include *mTableau* [13], *Pocket KRHyper* [2], *Delta* [14], and *Mini-ME* [15] reasoners [16]. In order to reuse as much as possible the existing work to optimize current Description Logics (DL) reasoners, we are more interested in another choice: reusing existing semantic reasoners on mobile devices. In particular, we have focused on porting and using existing DL reasoners on Android as they implement the latest reasoning algorithms and optimizations, and their codes have already been tested thoroughly. Moreover, as we will see later in this paper, the efforts needed to port some of them are indeed below the costs of developing new reasoners from scratch, as only a tiny fraction of the source code must be changed.

We have focused our research on devices using Android operating system [17] due to several reasons: 1) its diffusion (51% of devices use this operative system according to a recent estimation[1], but with a prevision of a heavy steady increase according to its shipment share of 78% during the first quarter of 2015[2]), 2) its openness and thorough documentation, and 3) the existence of a Java-like native virtual machine (Dalvik) that makes it easier to reuse existing Java applications, something very important since most of the semantic APIs and reasoners have been developed in this language.

In our previous works [18, 19], we began to study the reuse of existing Semantic Web APIs and DL reasoners on Android. The objective of this paper is to continue this line of research by studying more DL reasoners, performing a more complete evaluation of their performance, and providing a starting point for those who want to add semantics to their mobile systems.

In particular, the contributions of this paper are the following:

- We detail the level of support of some of the existing Semantic Web APIs and DL reasoners on Android devices. Moreover, we share our experience porting and using an important number of available reasoners (CB, ELK, HermiT, jcel, JFact, Pellet, and TrOWL), which makes it easier to port future versions, and might help porting other reasoners. Finally, we also high-

[1] `http://www.netmarketshare.com`, last accessed 2015-07-04.

[2] `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`, last accessed 2015-07-04.

light the difficulties that we have found in different versions of the same reasoners.

- We perform a complete empirical evaluation of the performance of the reasoners on Android devices and desktop computers. We consider two reasoning tasks (classification and consistency checking), analyzing the execution time and the impact of memory, virtual machine, and the OWL 2 profile (comparing the differences between the fragments OWL 2 DL and OWL 2 EL). We use more than 300 ontologies from the ORE 2013 ontology set.

The rest of this manuscript is organized as follows. Section 2 starts by overviewing some Semantic Web APIs and reasoners that will be considered throughout this paper and Section 3 summarizes our experiences porting these technologies to Android. Then, after describing our experimental setup in Section 4, Sections 5 and 6 evaluate the performance of the reasoners for the OWL 2 DL and OWL 2 EL profiles, respectively, while Section 7 studies the impact of the memory and the virtual machine. Next, Section 8 includes a global discussion. Finally, Section 9 overviews some related work and Section 10 sets out some conclusions and ideas for future work.

## 2. Overview of Semantic Web Technologies

In this section we overview the Semantic Web technologies that will be considered throughout this paper. In particular, we present the semantic APIs (Jena and OWL API) and DL reasoners that we tried to reuse on Android devices. For completeness sake, we will also reference other relevant DL reasoners in the literature. Finally, we review DL reasoners specifically designed for mobile devices.

### *2.1. Semantic Web APIs*

*Jena*[3] [20] is an ontology API to manage OWL ontologies and RDF data in Java applications. Jena is appropriate to manage OWL 1 Full ontologies, but support for OWL 2 is not available yet. However, it is much more used for the serialization of RDF triples and the manipulation of RDF graphs. Jena can interact with semantic reasoners to discover implicit knowledge. The latest versions of Jena are split into two packages, namely `jena-fuseki` (with the Jena SPARQL server), and `apache-jena` (with APIs, SPARQL engine, RDF database, and other tools).

*OWL API*[4] [21] is an ontology API to manage OWL 2 ontologies in Java applications and provides a common interface to interact with DL reasoners. It can be considered as a de facto standard, as the most recent versions of most of the semantics tools and reasoners use the OWL API to load and process OWL 2 ontologies. The OWL API is able to process each of the OWL 2 syntaxes defined in the W3C specification (functional, RDF/XML, OWL/XML, Manchester, and Turtle) and to identify the OWL 2 profiles (OWL 2 DL, OWL 2 EL, OWL 2 QL, and OWL 2 RL). The OWL API is less appropriate for the management of OWL 2 Full or RDF ontologies.

### *2.2. DL Reasoners*

*CB*[5] [22] (Consequence-Based) reasoner supports a fragment of OWL 2 (Horn-$\mathcal{SHIF}$). As its name suggests, the reasoner algorithm does not build models but infers new consequent axioms. CB is implemented in OCaml and, as far as we know, the only supported reasoning task is classification. It can be used from command line, as a Protégé plug-in, and through the OWL API.

*ELK*[6] [23], implemented in Java, is a Consequence-Based reasoner for a subset of OWL 2 EL. It supports different reasoning tasks, which include classification, consistency checking, subsumption, and realization. The classification procedure is different from other algorithms for OWL 2 EL. For instance, it includes several optimizations such as concurrency of the inference rules. ELK can be used through several interfaces, including OWL API.

*HermiT*[7] [24] implements a hypertableau reasoning algorithm with several optimization techniques. It supports OWL 2 and DL safe rules. Historically, it was the first DL reasoner that was able to classify some large ontologies (such as GALEN-original) thanks to a novel and efficient classification algorithm. Inference services include concept satisfiability, consistency, classification, subsumption, realization, and conjunctive query answering. HermiT is implemented in Java, and is accessible

[3] `http://jena.apache.org`

[4] `http://owlapi.sourceforge.net`

[5] `http://www.cs.ox.ac.uk/isg/tools/CB`

[6] `http://elk.semanticweb.org`

[7] `http://www.hermit-reasoner.com`

through several interfaces, including the OWL API and a Protégé plug-in.

*jcel*[8] [25] is a Java implementation of a tractable classification algorithm for a subset of OWL 2 EL. jcel is based on *CEL*[9] [26] (Classifier for $\mathcal{EL}$) reasoner, a Common LISP implementation of a rule-based completion classification algorithm. Both reasoners are open source and accessible through the OWL API; jcel can also be used using a Protégé plug-in.

*JFact*[10] is a Java port of the reasoner FaCT++, although it does not include all of its parts and provides an improved datatype support. FaCT++ reasoner[11] [27] is a successor of Fact reasoner (*FAst Classification of Terminologies*) [28]) using a different architecture and a more efficient implementation (FaCT was written in Common Lisp, and FaCT++ in C++). Both Fact++ and JFact completely support OWL 2 and implement a tableau algorithm [56] with several optimization techniques. Supported reasoning tasks include concept satisfiability, consistency, classification, and subsumption. From a historical point of view, FaCT++ was the first reasoner fully supporting OWL 2. Both reasoners can be used through the OWL API and are available under a GNU license.

*MORe*[12] [29] is an OWL 2 metareasoner that exploits module extraction techniques to divide complex reasoning tasks into simpler ones that can be solved using different reasoners. The modules of the ontology in the OWL 2 EL profile are solved by the ELK reasoner, and the more expressive ones are handled using HermiT and JFact. MORe currently supports classification and concept satisfiability. It is implemented in Java, open source, and accessible through the OWL API and using a Protégé plug-in.

*Pellet*[13] [30] supports full OWL 2 and DL safe rules. It implements a tableau algorithm with several optimization techniques. It was the first reasoner fully supporting OWL 1 DL. Inference services include concept satisfiability, consistency, classification, subsumption, realization, and conjunctive query answering. Pellet is implemented in Java and has multiple interfaces to access it, including OWL API.

*TrOWL*[14] [31] is implemented in Java and supports OWL 2, offering sound and complete reasoning for OWL 2 EL and OWL 2 QL, and approximate reasoning for OWL 2 DL. Inference services include classification and conjunctive query answering. TrOWL includes an OWL 2 EL reasoner (*REL*) to compute the classification and an OWL 2 QL reasoner (*Quill*) to answer conjunctive queries. Reasoning with OWL 2 DL ontologies is achieved by means of a syntactic approximation into OWL 2 EL or a semantic approximation into OWL 2 QL, depending on the reasoning task. TrOWL can be used through several interfaces, including OWL API.

*TReasoner* [15] [32] supports a subset of OWL 2, namely the Description Logic $\mathcal{SHOIQ}(\mathbf{D})$. *TReasoner* solves classification, concept satisfiability, and consistency using a tableau algorithm with several optimization techniques. It is implemented in C++ and supports the OWL API.

Table 1 shows a summary of every reasoner analyzed in this paper. There exist many others that we have not been able to consider yet. We will enumerate now, in alphabetical order, only those of them that will be mentioned during this paper: *BaseVISor*[16] [33], *Chainsaw*[17] [34], *ConDOR*[18] [35], *DB*[19] [36] *ELepHant*[20] [37], *fuzzyDL*[21] [38], *KAON2*[22] [39], *Konclude*[23] [40], *OWLIM*[24] [41] (a family of repositories including *SwiftOWLIM* reasoner), *Racer*[25] [42], *SHER* [43], *SOR* [44], *SnoRocket*[26] [45], *WSClassifier*[27] [46], and *WSReasoner*[28] [47].

### *2.3. DL Reasoners Designed for Mobile Devices*

Apart from the DL reasoners presented in the previous section, several DL reasoners were specifically designed to run on mobile devices. We dedicate this section to overview them in a chronologi-

[8] http://jcel.sourceforge.net
[9] http://lat.inf.tu-dresden.de/systems/cel
[10] http://jfact.sourceforge.net
[11] http://owl.man.ac.uk/factplusplus
[12] http://code.google.com/p/more-reasoner
[13] http://clarkparsia.com/pellet
[14] http://trowl.eu
[15] http://code.google.com/p/treasoner
[16] http://vistology.com/basevisor/basevisor.html
[17] http://sourceforge.net/projects/chainsaw
[18] http://code.google.com/p/condor-reasoner
[19] https://code.google.com/p/db-reasoner
[20] https://github.com/sertkaya/elephant-reasoner
[21] http://webdiis.unizar.es/~fbobillo/fuzzyDL
[22] http://kaon2.semanticweb.org
[23] http://www.derivo.de/en/produkte/konclude
[24] http://www.ontotext.com/owlim
[25] http://www.ifis.uni-luebeck.de/index.php?id=385
[26] http://github.com/aehrc/snorocket
[27] http://code.google.com/p/wsclassifier
[28] http://isew.cs.unb.ca/wsreasoner

| Reasoner | Profile | Language | License | OWL API |
|---|---|---|---|---|
| CB | OWL 2 DL (fragment) | OCaml | LGPL | Yes |
| ELK | OWL 2 EL (fragment) | Java | Apache 2.0 | Yes |
| HermiT | OWL 2 DL | Java | LGPL | Yes |
| jcel | OWL 2 EL (fragment) | Java | LGPL and Apache 2.0 | Yes |
| JFact | OWL 2 DL | Java | LGPL | Yes |
| MORe | OWL 2 DL | Java | GPL | Yes |
| Pellet | OWL 2 DL | Java | Dual | Yes |
| TReasoner | OWL 2 DL (fragment) | Java | GPL | Yes |
| TrOWL | OWL 2 DL (approximated) | Java | Dual | Yes |

Table 1: Semantic Web reasoners and some of their characteristics.

cal order. *Pocket KRHyper*[29] [2] was the first reasoning engine specifically designed for mobile devices. It can be seen as a version of the reasoner KRHyper [48] for devices with limited resources, thus disabling some of its original capabilities (such as default negation and term indexing). It is implemented in J2ME (Java Micro Edition) and implements a hypertableau algorithm for the DL $\mathcal{SHI}$. However, the reasoner suffers from scalability issues, as the authors state in [3].

Later on, Müller et al. [16] reported the implementation of tableau algorithm for mobile devices, introducing some optimizations to reduce the memory usage such as assigning natural numbers to concept expressions to reduce comparisons to integer operations. Their system is implemented in J2ME and supports the DL $\mathcal{ALCN}$ with unfoldable TBoxes, but it does not have a known name and is not publicly available.

*mTableau* [13, 49] is a modified version of Pellet 1.5 to work on mobile devices. The main idea is to introduce some novel optimization techniques, namely selective application of consistency rules, skipping disjunctions, and ranking of individuals and disjunctions leading to potential clashes. This reasoner is not publicly available.

*Delta* [14] is designed to be used on mobile devices, but no implementation details are given. The reasoner uses RDF to store the ABox and OWL RL to represent the TBox axioms. The main reasoning task is conjunctive query answering, which is solved by translating TBox axioms into rules to expand the RDF triple store. The reasoner uses incremental reasoning techniques to avoid recomputing all the inferences every time there is an update of the ABox facts. A preliminary evaluation is performed, obtaining sub-second query response times. This reasoner is not publicly available.

*Mini-ME*[30] [15](Mini Matchmaking Engine) is a mobile reasoner implemented from scratch. The supported DL is the DL $\mathcal{ALN}$, whereas the supported reasoning tasks are consistency, classification, concept satisfiability, subsumption, and other non-standard inference services (abduction, contraction, and covering). It is implemented in Java and can be run on Android devices as well as on desktop computers. Mini-ME can be accessed through the OWL API, as a OWLlink server, or using a Protégé plug-in. The authors have empirically compared the performance of Mini-ME in a mobile device and in a desktop computer. It turned out that reasoning times are roughly one order of magnitude higher in the Android device [15]. However, it should be stressed that these results only hold for the not very expressive logic $\mathcal{ALN}$, having polynomial computational complexity. The authors also performed some experiments proving that the Android version of Mini-ME outperforms an older version developed in J2ME [50].

## 3. Using Semantic Web APIs and Reasoners on Android

In this section, we overview some key features of Android and its current support for the semantic technologies we tested. Then, we move onto the details of our experience porting some of them which were not Android compatible.

### *3.1. Overview of Current Android Support*

Most of current popular semantic reasoners are implemented using Java (see Table 1) and are usually used along with semantic APIs (e.g., OWL

[29] `http://mobilereasoner.sourceforge.net`

[30] `http://sisinflab.poliba.it/swottools/minime`

API and Jena). While Android is a Linux-based operating system whose middleware, libraries, and APIs are written in C, it uses a Java-like virtual machine called Dalvik [51] that makes it possible to support Java code. In fact, Dalvik 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 completely align to Java SE and so it does not support J2ME classes, AWT or Swing. Thus, running semantic APIs and reasoners on Android could require some rewriting efforts.

Table 2 summarizes the current Android support for the main semantic APIs and DL reasoners. We tested all of them, and found out that *OWL API 3.4.10*, and the *jcel 0.19.1*, *JFact 0.9.1*, *TReasoner revision 22*, and *TrOWL 1.4* reasoners can be imported directly in Android projects. However, as the table shows, most of them (16 out of 21) are not directly compatible with Android. In the next section, we explain our experience trying to port these reasoners to Android.

| Software | Version | Originally compatible | Currently compatible |
|---|---|---|---|
| Jena | 2.12.0 | × | ✓ |
| OWL API | 3.4.10 | ✓ | ✓ |
| CB | build 12 | × | ✓* |
| CEL | 1.0 | × | × |
| Chainsaw | 1.0 | × | × |
| ConDOR | revision 13 | × | × |
| ELepHant | 0.4.0 | × | × |
| ELK | 0.4.0 | × | ✓* |
| FaCT++ | 1.6.3 | × | × |
| fuzzyDL | build 60 | × | × |
| HermiT | 1.3.8 | × | ✓* |
| jcel | 0.19.1 | ✓ | ✓ |
| JFact | 0.9.1 | ✓ | ✓ |
| KAON2 | unknown | × | × |
| Konclude | 0.6.0 | × | × |
| MORe | 0.1.5 | × | ✓* |
| Pellet | 2.3.1 | × | ✓* |
| Racer | 2.0 | × | × |
| TReasoner | revision 22 | ✓ | ✓ |
| TrOWL | 1.4 | ✓ | ✓ |
| WSClassifier | revision 1 | × | × |

*: It has been ported by us.

Table 2: Android support for some semantic APIs and DL reasoners.

### *3.2. Porting Semantic Technologies to Android*

As Table 2 showed, there are APIs and reasoners that do not work directly on Android. Thus, we tried to port them (or use alternative ones) to make them work in our Android projects. In the following, we detail how we have ported these different technologies, and we highlight some of the possible problems for those we did not successfully port.

As a summary, the main causes that we found for reasoners not to be directly imported in Android projects can be broadly classified as:

- Direct use of Java classes not supported in Android.
- Use of external libraries that use unsupported Java classes.

Most of these problems can only be detected at runtime. So, the process we followed consisted on importing each reasoner (downloaded from their websites) in an Android application that we developed to automate the testing, and running it. Then, for those reasoners that failed to run, we tried to detect the problematic classes/libraries by studying its code (whenever it was possible, as not all the developers made available the code of their reasoners). Finally, we tried to replace problematic classes/libraries by their equivalent ones for the Android platform. In this section, we explain the experience trying to port the technologies that failed to run on Android directly. Further and detailed information about the specific methods and classes changed for each reasoner can be found on the webpage of the project [12] together with, if the licenses make it possible, a download link.

We will firstly present the case of *Jena* API, then the successfully ported reasoners and, finally, the unsuccessfully ones. In both cases, reasoners will be presented in alphabetical order.

*Jena* cannot be directly imported into an Android project but there exists a project called Androjena[31] to port it to the Android platform. The latest version of Androjena 0.5, which was used in our tests, contains all the original code of Jena 2.6.2 and *can be used in Android projects.*

*CB* is implemented in OCaml and Android does not support it natively. There are some projects to develop OCaml interpreters for the platform, such as the OCaml Toplevel[32]; however, we chose a different approach: compiling the reasoner to native

[31] `https://code.google.com/p/androjena`

[32] `https://bitbucket.org/keigoi/ocaml-toplevel-android`

Android code. For that, we used the Android Native Development Kit (NDK)[33] to cross-compile the code for the ARM processor. *The resulting native code can be executed on Android using the command line tool Android Debug Bridge (adb).* To import this native code into an Android project, we could use the Java Native Interface (JNI) and Android NDK. However, for the purpose of this paper we tested the native code directly. Unfortunately, we have not already been able to access CB from the OWL API on Android.

*ELK* presents a problem with the only external library that the reasoner imports. `Log4j` is an open source (Apache License 2.0) logging utility that uses classes of the Java package `java.beans`, which is not completely supported in Android. There exists a port for this library[34] but in its current version it presents some problems. Therefore, the recommended process is to replace the Log4j library by `SLF4J`[35], which is supported in Android. In this case, we use the `log4j-over-slf4j` library that allows `log4j` applications to be migrated to `SLF4J` without changing the code. *With this replacement, ELK 0.4.0 can be used in Android projects.*

*HermiT* references unsupported Java classes (both in its source code and in the imported external library `JAutomata`). On the one hand, we detected problems with the debug, Protégé, and command line packages. Specifically, the references to `java.awt.point` and other Java AWT classes must be replaced as Android has its own graphical libraries. Due to their nature, we thought that these packages would not be required by a developer who uses the reasoner in an Android application, and therefore, we removed all of them from our port. On the other hand, `JAutomata`[36], a library for creating, manipulating, and displaying finite-state automata, presents two problems: 1) it references the aforementioned `java.awt.point` class in some hashing functions, and 2) it references two unsupported libraries, `JUnit`[37] and `dk.brics.automaton`[38]. To address the first problem, we just modified the hashing functions. For the second problem, on the one hand, we removed the `JUnit` library and its references (as it is a library used for development); and, on the other hand, we reimplemented part of `dk.brics.automaton`. This library is required as it contains a DFA/NFA (finite-state automata) implementation that is used to process datatypes of the ontology. It uses some files that contain automata that cannot be unmarshalled in Android (as they were marshalled using Java). Thus, to solve this problem, we reimplemented the marshalling/unmarshalling methods to create a new set of automata files compatible with Android. *With this changes HermiT 1.3.8 can be used in Android projects.*

*MORe* uses the HermiT, JFact, and ELK reasoners and, as explained previously, the original version of HermiT and ELK are not compatible with Android. Replacing the two reasoners by their ported versions fixes this problem. *Therefore, MORe 0.1.5 can be used in Android projects.*

*Pellet* presented problems with unsupported Java classes being referenced from its tests packages (that can be removed) and three external libraries: Jena (which can be replaced by Androjena as explained before), OWL API 2.2.0 (which can be removed from the final compiled version), and `JAXB`[39], a library to map Java classes to XML. `JAXB` uses the `javax.xml.bind` package and the `Xerces`[40] parser libraries which are not supported in Android. This latter problem can be solved by removing the `JAXB` .jar file and adding the source code of both `javax.xml.bind` and `Xerces` to our Android project. However, Dalvik has a limit of 65536 methods references per .dex file and it gets exceeded when applying this solution. To solve this, we removed the `JAXB` library and copied only the nine classes that Pellet needs from both the `java.xml.bind` package and the `Xerces` library to our Android project. *With this changes Pellet 2.3.1 can be used in Android projects.*

Apart from the above mentioned ones, we also tried other reasoners that we could not port to Android. Some reasoners were not available at the time when this work was performed, such as *SHER*, *SOR*, or *WSClassifier*.

*KAON2* presented problems when imported in an Android project. The source code of this reasoner is not available, so we have been able to detect possible problems only by analyzing the libraries it imports. Among them, the reasoner imports Java

[33] `http://developer.android.com/tools/sdk/ndk`
[34] `https://code.google.com/p/android-logging-log4j`
[35] `http://www.slf4j.org`
[36] `http://jautomata.sourceforge.net`
[37] `http://junit.org`
[38] `http://www.brics.dk/automaton`
[39] `https://jaxb.java.net`
[40] `http://xerces.apache.org`

Remote Method Invocation (RMI) which is not supported in Android. There are two projects to port this library to Android, `LipeRMI`[41] and `RipeRMI`[42], which we have used in other projects and are a possible replacement for RMI. However, they do not align completely with the API of RMI, so, modifying the code would be necessary. The reasoner might also need further code rewriting that could only be detected by accessing the source code. *Therefore, the latest version of KAON2 cannot be used in Android projects.*

The fuzzy ontology reasoner *fuzzyDL* uses the `Gurobi`[43] library, an optimization programming solver. This library is not supported in Android and, up to the authors' knowledge, has not a supported replacement. In addition, the library has a proprietary license and so, we could not explore the steps needed to port it to Android. *Therefore, fuzzyDL build 60 cannot be used in Android projects.*

Finally, other reasoners are developed in languages different from Java that Android does not support natively. In these cases, one could try the same approach presented for *CB*: compiling the code for ARM with the help of the Android NDK and using JNI to import the code from an Android project. For example, there are a lot of reasoners implemented in C++, such as *Chainsaw*, *ConDOR*, *FaCT++*, *ELepHant*, *Konclude*, and *WSClassifier*. Porting these C++ reasoners would also make it easier to support some metareasoners written in Java but using reasoners implemented in C++. For example, *Chainsaw* uses *FaCT++* and *WSClassifier* uses *ConDOR*. There are also some reasoners implemented in Lisp, such as *Racer* and *CEL*. Note that there is also a *Racer* server version that could be used on an external device and used from clients. However, that would require a connection between the mobile device (client) and the server which defeats the purpose of our work.

## 4. Experimental Setup

This section describes the experimental setup used in our empirical evaluation.

### *4.1. Selecting the Ontology Dataset*

To evaluate the performance of the studied reasoners, we selected the ORE 2013 ontology set [52] which contains 200 ontologies per profile (i.e., OWL 2 EL, OWL 2 RL, and OWL 2 DL) from the NCBO BioPortal[44], the Oxford Ontology Library[45], and the Manchester Ontology Repository[46]. Every ontology has at least 100 logical axioms and 10 named concepts, and they are classified according to their number of logical axioms as *small* ($\leq$ 500), *medium* (between 500 and 4999), and *large* ontologies ($\geq$ 5000). The ORE 2014 ontology set, with 16555 ontologies, is too large for our purposes [53].

In our case, we focused on the OWL 2 DL and OWL 2 EL ontology sets. We did not select the OWL 2 RL and OWL 2 QL profiles as we were not able to port any reasoner specifically for these profiles. Besides, we had to take into account the restrictions that mobile devices suffer from, especially the limited CPU, memory, and battery. For this reason, we selected a subset of the ORE 2013 ontology set carrying out the following steps:

1. We ordered the ontologies according to the size of the file as, when evaluating mobile devices, we should not only take into account the number of logical axioms, but also the file size (which is directly related to the memory needed to load such ontology). We could not ignore annotation axioms as they were problematic in our scenario: they also need to be processed by the ontology parser and, thus, they might consume extra memory temporally (e.g., by the OWL API parsers).
   Please note that the file size is just a heuristic, because it depends on the OWL 2 syntax, the length of URIs, and other non-logical aspects. In our experiments, we have considered OWL/XML syntax[47].
2. The maximum heap size per application provided by the Android version in the test devices is 256 MB (after setting the variable `android:largeHeap=''true''` to get the maximum heap size for a mobile application). This could be the theoretical maximum size of an ontology loaded on Android, but we must build instances of the OWL API class `OWL-Reasoner` in the device's memory. So, we filtered the ontology sets to take out the ontologies whose files occupied more than 128 MB.

[41] `http://lipermi.sourceforge.net`
[42] `https://code.google.com/p/ripermi`
[43] `http://www.gurobi.com`
[44] `http://bioportal.bioontology.org`
[45] `http://www.cs.ox.ac.uk/isg/ontologies`
[46] `http://rpc295.cs.man.ac.uk:8080/repository`
[47] `http://www.w3.org/TR/owl-xmlsyntax`

This resulted in 186 OWL 2 DL and 194 OWL 2 EL ontologies. However, four of the OWL 2 DL ontologies and one OWL 2 EL ontology were not admitted by our testing application because of their URIs. Therefore, our final *DL ontology set* used in our experiments contains 182 OWL 2 DL ontologies, distributed as follows: 43 small, 103 medium, and 36 large ones; whereas our *EL ontology set* has 193 ontologies: 72 small, 82 medium, and 39 large ones. Our full ontology set and some ontology stats (such as their size, number of axioms, and expressivity) can be found at [12].

### *4.2. Selecting the Devices*

We considered two mobile devices for the tests: a smartphone and a tablet. The smartphone selected was a Galaxy Nexus (Android 4.2.1, 1.2 GHz dual-core, 1 GB RAM, released in 2011, in the following denoted as A1), and the tablet was a Galaxy Tab 2 7.0 (Android 4.1.2, 1 GHz dual-core, 1 GB RAM, released in 2012, denoted A2). In order to avoid battery shortage, both devices were plugged in during all the tests.

In our previous work [18], we also performed some preliminary tests with a Galaxy Tab tablet (Android 2.3.3, 1.0 GHz single-core, 512MB RAM, released in 2010, denoted as A3) using five popular ontologies: *Pizza*[48] and *Wine*[49], which are two expressive ontologies; *DBpedia* 3.8[50] (TBox), which can be useful for mobile application developers to access the structured content of DBpedia (a semantic entry point to Wikipedia) [55]; and the Gene Ontology (*GO*)[51] and the US National Cancer Institute (*NCI*)[52] ontologies, which contain a high number of concepts. Table 3 shows a summary of the results obtained when comparing the performance of the devices A1 and A3. A1 outperformed A3 up to a 30% of increment on the performance in some situations, and thus, we left it aside for our experiments.

According to the official Android report[53], and as of July 2015, the use of the Jelly Bean version of Android (from 4.1.x to 4.3) represents around 37.4% of current Android devices (76.6% considering also 4.4 devices as the core of the OS is almost the same). Also, as of 2015, most of the devices on the market have similar or even better capabilities than the Galaxy Nexus and the Galaxy Tab 2. Thus, with the Galaxy Nexus and the Galaxy Tab 2 we are representing the average current Android device in terms of capabilities and Android version.

| | | HermiT | JFact | Pellet |
|---|---|---|---|---|
| DBpedia | A1 | 5.13 | UDT | 63.15 |
| | A3 | 8.87 | UDT | 115.30 |
| GO | A1 | 487.98 | 435.60 | 83.97 |
| | A3 | OOM | OOM | OOM |
| NCI | A1 | 2020.48 | OOM | OOM |
| | A3 | OOM | OOM | OOM |
| Pizza | A1 | 10.43 | 3.42 | 20.77 |
| | A3 | 14.88 | 4.90 | 33.22 |
| Wine | A1 | 361.38 | 1609.32 | 131.80 |
| | A3 | 511.97 | 2196.05 | 194.12 |

Table 3: Comparison of classification time (in seconds) for two Android devices. *OOM*: Out Of Memory; *UDT*: Unsupported Data Type.

Also, we considered a desktop computer to determine how slow is reasoning on Android compared to this baseline (taking into account that the desktop computer hardware outperforms Android devices, and that their virtual machines are optimized for different purposes). In this case the desktop computer selected (denoted PC) was a Windows 64-bits, i5-2320 3.00 GHz, 16 GB RAM (12 GB were allocated for the JVM in the tests).

### *4.3. Selecting the Tasks*

We analyzed the behavior of the reasoners for two standard Description Logic inference services that were part of the ORE 2013 competition:

- *Ontology classification*: computing the complete class hierarchy based on the subsumption relation between the ontology classes.
- *Ontology consistency*: checking whether an ontology contains any contradictions or not.

For the moment, we did not consider other tasks such as concept satisfiability[54], query answering,

[48] `http://www.co-ode.org/ontologies/pizza/pizza.owl`
[49] `http://www.w3.org/TR/owl-guide/wine.rdf`
[50] `http://dbpedia.org/Ontology`
[51] `http://www.geneontology.org`
[52] `http://ncit.nci.nih.gov`
[53] `http://developer.android.com/about/dashboards/index.html`, last accessed 2015-07-04.

[54] Concept satisfiability is subsumed by classification, so one should test concept satisfiability on non-classified ontologies.

or realization[55] because the results obtained are strongly dependent on the particular choice of the selected concept, query, or individual, respectively. To obtain significant results, each test would have to be repeated for a considerable amount of different elements on each ontology. Moreover, the selected tasks are being currently used in semantic mobile applications. For example, SHERLOCK [4], FaceBlock [11], and Rafiki [9] use classification, whereas Triveni [57] also checks consistency.

We measured the performance of the different reasoners and devices for these tasks in terms of finished tasks and computation time. Due to the high variance of processing time on Android devices observed in our preliminary experiments [18, 19], we repeated every test three times and computed the average and variance of the processing time. We considered a task as finished if it was processed without throwing any error and within a defined timeout, which was set to 25 minutes for Android devices and 5 minutes for the desktop computer.

Regarding the consumed memory, on Android it is difficult to obtain a precise measure of the memory consumed (the most accurate value would be the so called Proportional Set Size, which includes the private memory and divides the shared pages between all the processes that share them, but it is just an estimation); thus, in our experiments, we made the biggest amount of memory available (setting the heap value), and focused on the amount of tasks finished with that heap size as limit for all the reasoners. Moreover, we considered measuring battery consumption as well, but we found several difficulties to do that accurately on Android 4.x devices. As shown in [58], where the power consumption for some reasoners on Android 4.x is analyzed, measuring this factor requires external hardware connected to the battery of the device.

Finally, note that we are not checking if the result returned by the reasoner is correct, we only guarantee that the results are the same on Android devices and on the desktop computer. The ORE 2013 competition estimates the correctness of the DL reasoners by a majority vote and publishes their results, so the interested reader is referred to [52].

[55] As defined in [56], here, we consider realization as finding the most specific concepts a given individual is an instance of.

### *4.4. Selecting the Reasoners*

In our previous work, we detected that reasoning on Android devices was up to 100 times slower than on a desktop computer for certain ontologies and reasoners [19]. Therefore, testing all the reasoners would require hundreds of computation hours, and so, we selected a representative subset of popular reasoners for our tests: *ELK*, *HermiT*, *jcel*, *JFact*, *Pellet*, and *TROWL*. *MORe* was not included in our tests because it uses *ELK*, *HermiT*, and *JFact* that are already being analyzed.

Moreover, we have not included CB reasoner in the experiments with the complete ontology set, despite having tested it in our preliminary work [18]. Although CB has the advantage of running outside the virtual machine using native code, and thus not being subject to the restrictions imposed by the runtime environment, its expressivity and ease of use are problematic. CB requires to work with Horn-$\mathcal{SHIF}$ ontologies, but only 12% of our DL ontology set are Horn-$\mathcal{SHIF}$ ontologies. Furthermore, using CB in a mobile application might be quite complex due to different steps needed to make the port work. This might be a barrier to its use in mobile applications, especially when there are other reasoners than can be imported and used almost directly on Android. Indeed, we have not been able yet to access CB on Android using the OWL API, which has become a de facto standard. Thus, we decided to restrict the tests to the reasoners accessible using the OWL API, as they are more prone to be used in practice.

Since CB executes native code outside the virtual machine of Android, it is not subject to the restrictions imposed by the virtual machine, while the other reasoners run within the virtual machine in a more constrained framework. It is worth to include some results showing the effect of this. Table 4 compares the classification times (in seconds) of three popular Horn ontologies (DBpedia, GO, and NCI) on the desktop computer and the smartphone using CB and ELK. In this case, since CB is not accessed using the OWL API, ELK is accessed using its own API to avoid a possible overhead caused by the OWL API. DBpedia includes datatypes and thus is not fully supported by CB, so we do not consider the results of the classification[56]. Notice how, on PC, CB is 2.3 times faster

[56] Note that [18] considers the results because the results of the classification happen to be correct.

than ELK for the GO ontology, but on the Android device CB is 3.5 faster than ELK. Furthermore, on PC, ELK is slightly faster than CB for the NCI ontology, but on A1 CB is 8 times faster. Therefore, running native code directly without using the virtual machine on Android devices seems to make a difference. A more detailed study of this fact is left as future work.

| | | CB | ELK |
|---|---|---|---|
| DBpedia | PC | UDT | 0.5 |
| | A1 | UDT | 19.7 |
| GO | PC | 0.6 | 1.4 |
| | A1 | 8.6 | 30.6 |
| NCI | PC | 2.6 | 2.5 |
| | A1 | 35.4 | 200.7 |

Table 4: Comparison of classification time (in seconds) for PC and Android. *UDT*: Unsupported Data Type.

Regarding the reasoners designed for mobile devices (see Section 2.3), we were forced to discard *Mini-ME*, the only reasoner that is available and compatible with Android. Before starting the experiments, we compared the expressivity of the ontologies in both ontology sets with the expressivity that Mini-ME supported, and, at first, 160 ontologies from the DL ontology set and 42 from the EL ontology set lay out from the expressivity supported ($\mathcal{ALCN}$). Anyway, we computed the classification of the EL ontology set, but 96 ontologies did not finish due to several problems (such as class exceptions), and 97 ontologies reached the established timeout for the tests. Thus, the results of 9 ontologies (5 %) are not significant enough to compare *Mini-ME* with the other reasoners.

### *4.5. Automating the Tests*

Given the large number of ontology-reasoner tests to perform on each device (more than 11.000 tests per device: 182 DL ontologies and 4 reasoners, 193 EL ontologies and 6 reasoners, two tasks per ontology and reasoner, and three repetitions for each task), we needed a tool to automate the process as much as possible. We studied the possibility of automating the tests via scripting using the Android Debug Bridge (adb)[57] interface, but, we decided to perform the tests in a scenario closer to what developers using the reasoners will face.

For that, we created an Android application that enables us to select the reasoner to test, the ontology set, and the reasoning task, and automatically performs the assigned task. This application uses the OWL API 3.4.10 to load each ontology, and measures the time since the creation of the reasoner (`ReasonerFactory.createReasoner(ontology)`). After each test, the application stores the results (elapsed time, result of the task, and information about the experimental setup) in an SQLite database.

The application is composed of three main elements: the GUI (an Android Activity) to configure the test to be performed, and two Android services: one to manage the automation of each subtask, and another one to perform the actual reasoning task. As advised in the Android development documentation, long term operations should be always encapsulated as standalone services (instead of background threads/tasks). The reasoning task is subject to a predetermined timeout to control the execution times. At first, we implemented the mobile application using the default behaviour of Android, this is, just one system process hosts all the different threads and services of the application. This implies that the heap space also is unique and shared among all the elements of the application (activities and services). Moreover, we relied on Android's mechanism to relaunch the application if it was selected to be killed (e.g., due to excessive resource consumption).

However, when Android detected that a task consumed too much resources, it killed the whole application instead of killing only the current task. Despite the relaunching mechanisms, as the application was selected to be preempted due to its intensive use of resources, it seemed to be penalized and was not relaunched immediately (in fact, it is sensible to do so: a task that has been killed due to greedy use of resources may retry again if relaunched under similar conditions). Therefore, waiting for the application to be relaunched increased heavily the times needed to obtain results. Moreover, we had to implement ad hoc code to recover from these situations, e.g, via checkpointing.

To avoid this situation, we made all the components independent by giving them their own process (with their own virtual machine instance and heap memory), and, instead of relying on Android, we established an external timeout (handled by the

[57] `http://developer.android.com/tools/help/adb.html`

manager service) that acts as a watchdog for the times that the reasoning task is killed externally. This way, Android only killed the service that was performing the reasoning task, allowing the manager service to discard the test when any of the two timeouts for it were reached (the execution time timeout and the watchdog). This saved a lot of time, and gave us a valuable Android lesson: to move intensive processing tasks to a service (not an AsyncTask, but a complete Service) with its own process in order to protect the main application.

### *4.6. Verifying the Android Versions*

Before leaping into the main experiments, we performed a set of tests on the selected reasoners to test whether they produced exactly the same results on Android devices as on desktop computers. The aim of these tests was twofold: on the one hand, we wanted to check the behavior of the Android-compatible libraries we used to replace the unsupported ones; and on the other hand, we wanted to check the behavior of the reasoners that can be directly imported in Android projects.

We focused on the classification task because the consistency checking is just a yes-no question. First, as baseline, we obtained the sets of subsumption axioms computed by the original version of the reasoners running on the desktop computer. Then, we checked that the results obtained by the reasoners running on Android devices contained exactly the same axioms (of course, this was done separately per ontology). The only mismatches we found on our selected reasoners were due to two different reasons: sometimes the reasoning task on Android did not end before the timeout (and therefore, no comparison could be made), or there were problems with some ontologies due to the encoding of the ontology files, which led to malformed URIs (these problems disappeared once we aligned the file encodings, as detailed in Section 8).

Note that verifying the Android versions is indeed necessary: we found some examples where different versions of the ported reasoners gave different results on Android devices and in the desktop computer[58]. Thus, once we tested that we had not introduced any problem when porting the reasoners to Android, we moved on to the performance experiments.

In the following sections, we present our analysis of the performance of reasoners on mobile devices grouped by ontology profile. First, we present the results obtained for the OWL 2 DL profile and then for the OWL 2 EL profile (the same order used in the ORE 2013 report [52]) in Sections 5 and 6, respectively. For each profile, we present the results obtained when comparing the reasoners in terms of finished tasks and average computation time needed per task. Then, we compare the time consumption of the reasoners with the minimum set of ontologies that every reasoner and device was able to process. Next, in Section 7 we study the role of the limitation of memory and the virtual machine on the reasoning on Android devices.

## 5. Comparing the Reasoners for the OWL 2 DL Profile

In this section, we detail the results of our performance experiments for the OWL 2 DL profile. For our DL ontology set (with 182 ontologies), we tested the following reasoners: *JFact 0.9.1*, *HermiT 1.3.8*, *Pellet 2.3.1*, and *TrOWL 1.4*. As already mentioned, we first detail the number of finished tasks per reasoner and device, to then move on to their performance.

### *5.1. Comparing the Number of Finished Tasks*

The results for the classification task for the desktop computer (PC), the Galaxy Nexus (A1), and the Galaxy Tab 2 (A2) are shown in Figure 1(a), Figure 1(b), and Figure 1(c), respectively; for the consistency checking results for PC, A1, and A2 see Figure 1(d), Figure 1(e), and Figure 1(f).

First of all, analyzing the results in terms of finished tasks, we can observe that, as expected, the reasoners on PC finished more tasks than on Android due to the more powerful hardware. Only *HermiT* and *Pellet* finished a similar number of tasks on the desktop computer and on Android devices when checking the consistency (181 vs 180 and 177 vs 175-176). Moreover, the reasoners on A1 finished more tasks than the same reasoners running

[58] In JFact 1.2.1, the results of the classification and the consistency checking are different on PC and Android. For example, the ontology `aee636cb-4238-41af-a3d6-541d30f2e7ed_spills.owl` is correctly identified as consistent by the PC version, but is inconsistent according to the Android version. Furthermore, in the ontology `52cf3ab5-1662-4296-835e-b22ac92339e7_.2_DUL.owl`, the Android version misses two subclasses of the class `Role`, namely `Entity` and `E1.CRM_Entity`. This problem does not happen in JFact 0.9.1.

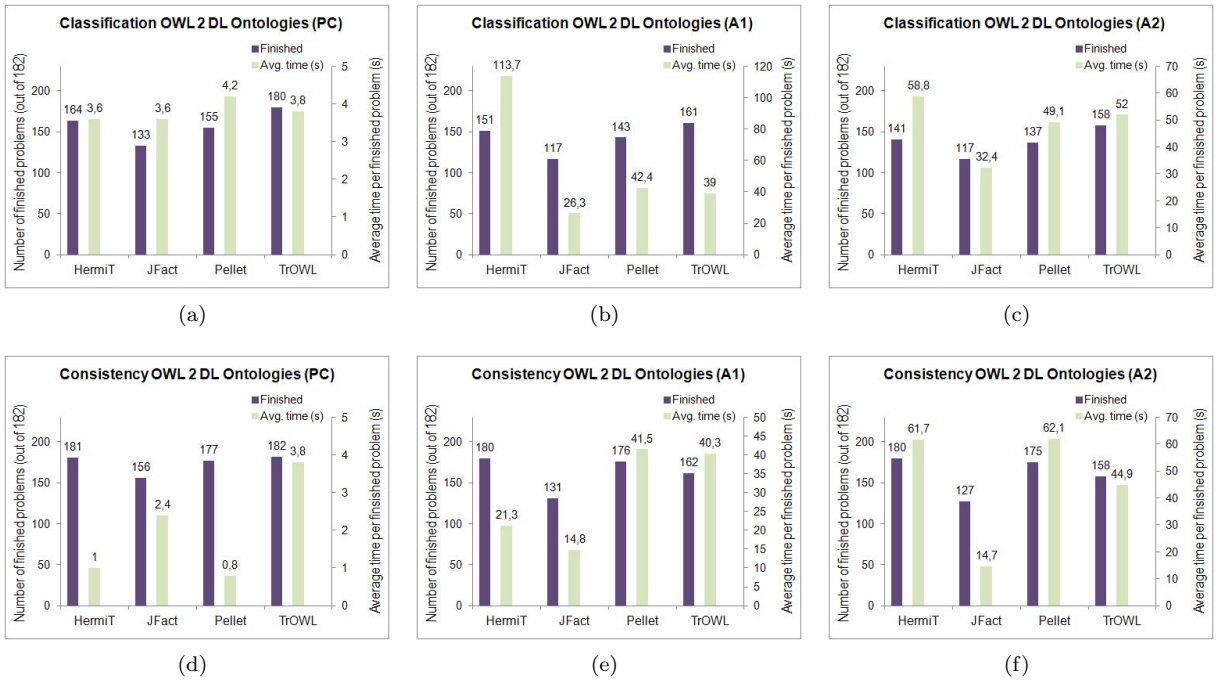



Figure 1: Results (finished tasks/average time) for the complete OWL 2 DL ontology set.

on A2 for most of the tests. There are only two situations where both devices finished the same number of tasks: computing the classification with *JFact* (117 out of 182 tasks on both devices), and checking the consistency with *HermiT* (180 out of 182 tasks on both devices).

Although the number of finished tasks on the desktop computer and on the mobile devices are different, both Android devices follow the PC trend for the classification task: *TrOWL* is the reasoner that finished a higher number of tasks, followed by *Hermit*, *Pellet*, and *JFact*. For the consistency checking, the PC trend is not followed by the Android devices: on the Android devices both *HermiT* and *Pellet* finished more tasks than *TrOWL* while *TrOWL* was the reasoner that finished more tasks on PC.

Analyzing the results of *TrOWL* for the consistency checking on A1, we noticed that the timeout period elapsed for 13 tasks, while it never elapsed for any task on PC. Also, 7 large ontologies that were processed by TrOWL on PC threw out of memory errors on the smartphone. Table 5 shows the number of tasks that could not been finished by the reasoners on each device and by each reasoner. We classify the reasons for not finishing as elapsed time out ("T/O" column in the table), and others ("Other" column). Notice that, as PC had enough memory to perform the reasoning, the errors on the "Other" column for PC are mostly due to unsupported data types. On the Android devices, the same errors always occur and some additional problems appear: usually, out of memory issues when processing large ontologies. Notice also that, in some situations, timeout problems on PC are translated into out of memory problems on Android devices. This happens because the timeout of PC was set to 5 minutes while on Android it was set to 25 minutes. Therefore, for some complicated tasks the reasoner run out of memory before the timeout elapsed (for example, *HermiT* in the consistency checking).

We would like to highlight that, although the results obtained for this test (Figure 1) enable us to compare the reasoners in terms of number of finished tasks, one should be cautious when comparing their processing times. The most difficult ontologies (i.e., the most expressive or large ones) require more time to be classified and so, completing more tasks

| Reasoner | Classification | | Consistency | |
|---|---|---|---|---|
| | T/O | Other | T/O | Other |
| HermiT(PC) | 13 (7.1%) | 5 (2.7%) | 1 (0.6%) | 0 (0%) |
| HermiT(A1) | 20 (11%) | 11 (6%) | 0 (0%) | 2 (1.1%) |
| HermiT(A2) | 28 (15.4%) | 13 (7.1%) | 0 (0%) | 2 (1.1%) |
| JFact(PC) | 18 (9.9%) | 31 (17%) | 8 (4.4%) | 18 (9.9%) |
| JFact(A1) | 26 (14.3%) | 39 (21.4%) | 18 (9.9%) | 33 (18.1%) |
| JFact(A2) | 23 (12.6%) | 42 (23.1%) | 24 (13.2%) | 31 (17%) |
| Pellet(PC) | 20 (11%) | 7 (3.9%) | 5 (2.7%) | 0 (0%) |
| Pellet(A1) | 26 (14.3%) | 13 (7.1%) | 6 (3.3%) | 0 (0%) |
| Pellet(A2) | 33 (18.1%) | 12 (6.6%) | 7 (4%) | 0 (0%) |
| TrOWL(PC) | 1 (0.6%) | 1 (0.6%) | 0 (0%) | 0 (0%) |
| TrOWL(A1) | 9 (5%) | 12 (6.6%) | 13 (7.1%) | 7 (3.9%) |
| TrOWL(A2) | 18 (9.9%) | 6 (3.3%) | 18 (9.9%) | 6 (3.3%) |

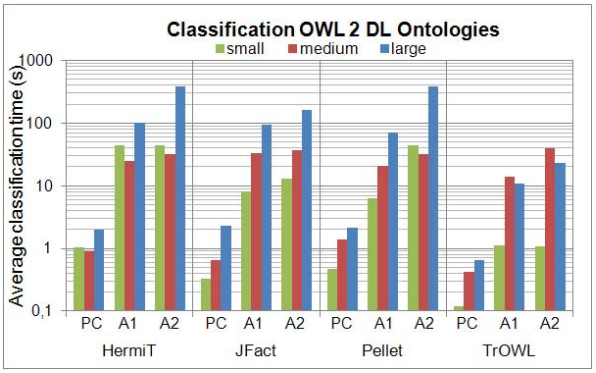
Table 5: Errors for uncompleted tasks in the DL ontology set.

could imply increasing the average time per finished problem. For example, notice that HermiT running on A1 completes 151 classification tasks with an average time per task of 113.7s while the same reasoner running on A2 completes 141 tasks with an average time of 58.8s (the average time increases about 50 seconds because of these 10 more challenging tasks). Therefore, we performed the following test to be able to compare the processing time of the reasoners.

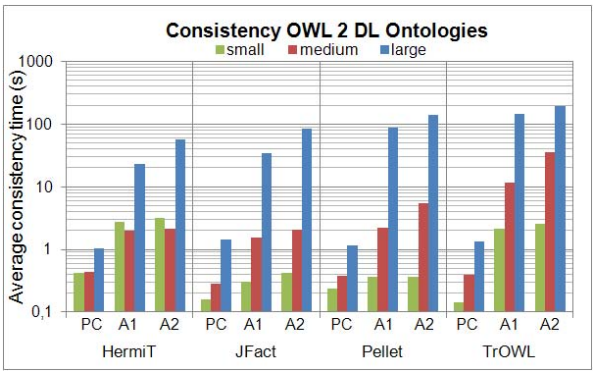
### 5.2. Comparing the Processing Time

To fairly compare the reasoners regarding the average processing time needed per ontology, we selected the minimum set of DL ontologies that all the devices and all the reasoners were able to process. We also split the ontologies with respect to their number of axioms into three subsets (small, medium, and large) obtaining a *minimum DL set* of 93 ontologies for the classification task (29 small, 62 medium, and 2 large ontologies), and 124 ontologies for the ontology consistency checking task (37 small, 81 medium, and 6 large ontologies).

*Comparing Trends.* Figure 2(a) and Figure 2(b) show the results obtained for the reasoners computing the classification and consistency, respectively, on the three devices with the minimum set of DL ontologies. First of all, as in our previous test, the mobile devices follow the PC general trend. There are two exceptions: on the one hand, *TrOWL* was slightly faster on PC for the classification of the medium set of ontologies than for the large one (about 0.2s), but on the mobile devices it was slightly slower (around 3s on the A1 device); on the other hand, *HermiT* was slightly faster on PC for checking the consistency of the small set than for the medium one (around 0.02s), but, on the mobile devices, it was slightly slower (around 0.7s on the A1 device). However, in the case of the classification in *HermiT*, the difference between the small and medium sets is also small but the trend on the desktop computer and the mobile devices is similar.




(a)




(b)

Figure 2: Average computing time for each ontology group in the minimum set of OWL 2 DL ontologies processed by all the devices and reasoners.

*Comparing Performance.* Table 6 shows the difference on the performance of the desktop computer, the smartphone, and the tablet in terms of the number of times of PC being faster than the Android devices. In general, PC outperformed A1 and A2 for all the reasoners. The greatest differences happen in the large ontology set, where, for example, the desktop computer is almost 110 times faster than A1 when checking consistency using *TrOWL*, and almost 200 times faster than A2 when computing classification with *HermiT*. However, the consistency checking of the small set of ontologies in the PC was "only" 2 times faster

than A1 for *JFact* (from 0.16s to 0.31s) and *Pellet* (from 0.24s to 0.37s).

| | | Classification | | | Consistency | | |
|---|---|---|---|---|---|---|---|
| | | S | M | L | S | M | L |
| HermiT | A1 | 44 | 28 | 52 | 7 | 5 | 22 |
| | A2 | 44 | 36 | 193 | 8 | 5 | 54 |
| JFact | A1 | 25 | 51 | 41 | 2 | 5 | 24 |
| | A2 | 40 | 57 | 71 | 3 | 7 | 59 |
| Pellet | A1 | 14 | 15 | 34 | 2 | 6 | 76 |
| | A2 | 21 | 46 | 38 | 2 | 14 | 123 |
| TrOWL | A1 | 9 | 33 | 16 | 15 | 29 | 109 |
| | A2 | 9 | 96 | 35 | 18 | 90 | 142 |

Table 6: Number of times (rounded to the closest integer) of the PC version being faster than the Android ones for the small (S), medium (M), and large (L) OWL 2 DL ontology set.

*TrOWL* was the fastest reasoner in both devices for the classification of all the ontology sets, but it uses approximate reasoning in OWL 2 DL. Note also that the difference between the different reasoners is smaller on PC than on mobile devices, for both reasoning tasks. In the mobile devices, the order of the reasoners according to their reasoning times is usually the same as in the desktop computer, although this is not always the case due to the variance of the results obtained for each test repetition.

## 6. Comparing the Reasoners for the OWL 2 EL Profile

In this section, we detail the results of our performance experiments for the OWL 2 EL profile. For the OWL 2 EL ontology set (with 193 ontologies), we tested the following reasoners: *ELK 0.4.0*, *HermiT 1.3.8*, *jcel 0.19.1*, *JFact 0.9.1*, *Pellet 2.3.1*, and *TrOWL 1.4*.

### *6.1. Comparing the Number of Finished Tasks*

The results for the classification task for the desktop computer (PC), the Galaxy Nexus (A1), and the Galaxy Tab 2 (A2) are shown in Figure 3(a), Figure 3(b), and Figure 3(c), respectively; for the consistency checking results for PC, A1, and A2 see Figure 3(d), Figure 3(e), and Figure 3(f).

As above mentioned for the DL ontology set, we can observe that the reasoners on PC finished more tasks than on Android. However, the difference on the number of finished tasks is smaller than for the DL ontology set. On the one hand, in the DL ontology set, the difference of finished tasks on PC and A1 is: 16 (for classification) and 25 (for consistency checking) for *JFact*; 19 and 20 for *TrOWL*; 13 and 1 for *HermiT*; and 12 and 1 for *Pellet*. On the other hand, in the EL ontology set the difference of finished tasks on PC and A1 is: 12 and 10 for *JFact*; 2 and 2 for *TrOWL*; 4 and 4 for *jcel*; 4 and 1 for *HermiT*; 4 and 1 for *Pellet*; and 1 and 0 for *ELK*. This can be explained because of the difference of expressivity of the two ontology sets. As the EL profile is less expressive, performing reasoning tasks within this profile is less costly. Therefore, more tasks can be finished on the mobile devices.

Also, in general, A1 finished more tasks than A2. There are only two situations where both devices finished the same number of tasks: first, *ELK* finished 190 out of 193 classifications on both devices; and second, *HermiT* finished 192 out of 193 consistency checkings on both devices.

Table 7 shows the results for the analysis of the tasks that were not finished. As for the DL ontology set, the errors of the "Other" column for the PC are mostly due to unsupported datatypes or ontologies that could not be processed by the reasoner. On the Android devices, the increment on the number of "Other" errors is due to out of memory errors. Notice also that the number of "Other" errors for the consistency checking is zero in *HermiT*, *Pellet*, and *ELK*, for the three tested devices.

| Reasoner | Classification (193) | | Consistency (193) | |
|---|---|---|---|---|
| | T/O | Other | T/O | Other |
| ELK(PC) | 2 (1%) | 0 | 0 | 0 |
| ELK(A1) | 1 (0.5%) | 2 (1%) | 0 | 0 |
| ELK(A2) | 2 (1%) | 1 (0.5%) | 1 (0.5%) | 0 |
| HermiT(PC) | 3 (1.6%) | 0 | 0 | 0 |
| HermiT(A1) | 7 (3.6%) | 0 | 1 (0.5%) | 0 |
| HermiT(A2) | 10 (5.2%) | 0 | 1 (0.5%) | 0 |
| jcel(PC) | 1 (0.54%) | 13 (6.7%) | 0 | 13 (6.7%) |
| jcel(A1) | 0 | 18 (9.3%) | 5 (2.6%) | 12 (6.2%) |
| jcel(A2) | 7 (3.6%) | 12 (6.2%) | 7 (3.6%) | 13 (6.7%) |
| JFact(PC) | 6 (3.1%) | 2 (1%) | 3 (1.6%) | 1 (0.5%) |
| JFact(A1) | 12 (6.2%) | 8 (4.1%) | 8 (4.1%) | 6 (3.1%) |
| JFact(A2) | 10 (5.2%) | 12 (6.2%) | 9 (4.7%) | 7 (3.6%) |
| Pellet(PC) | 3 (1.6%) | 4 (2.1%) | 0 | 0 |
| Pellet(A1) | 6 (3.1%) | 5 (2.6%) | 1 (0.5%) | 0 |
| Pellet(A2) | 10 (5.2%) | 5 (2.6%) | 3 (1.6%) | 0 |
| TrOWL(PC) | 2 (1%) | 0 | 1 (0.5%) | 0 |
| TrOWL(A1) | 2 (1%) | 2 (1%) | 2 (1%) | 1 (0.5%) |
| TrOWL(A2) | 5 (2.6%) | 1 (0.5%) | 4 (2.1%) | 1 (0.5%) |

Table 7: Errors for uncompleted tasks in the EL ontology set.

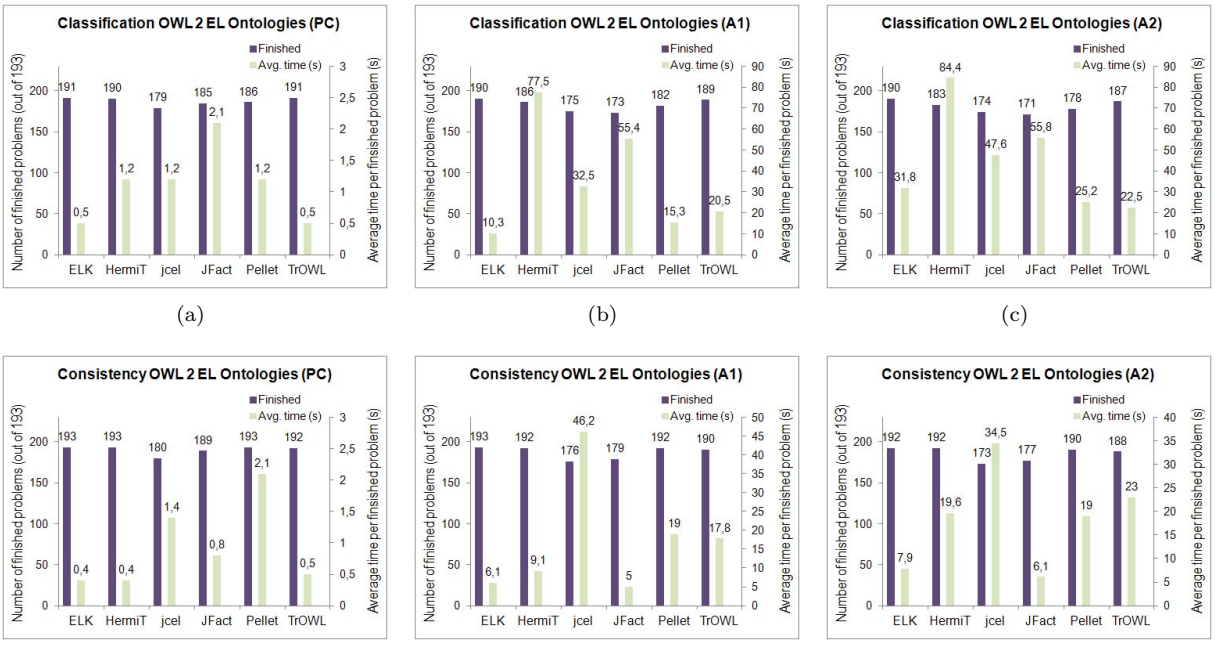



Figure 3: Results (finished tasks/average time) for the complete OWL 2 EL ontology set.

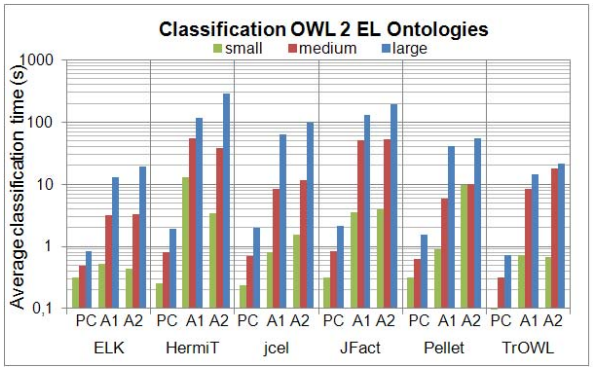
### *6.2. Comparing the Processing Time*

As we did for the DL ontology set, we selected the minimum set of EL ontologies that all the devices and all the reasoners were able to process to be able to fairly compare the performance of the reasoners regarding the average processing time per ontology. We also split the ontologies with respect to their number of axioms (small, medium, and large) obtaining a *minimum EL set* of 152 ontologies for classification (62 small, 67 medium, and 23 large ontologies) and 166 ontologies for consistency checking (65 small, 72 medium, and 29 large ontologies).

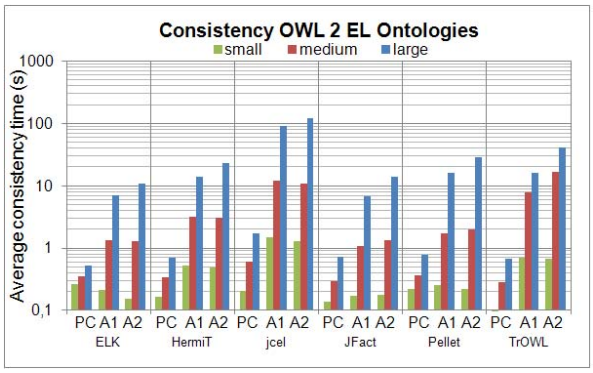
*Comparing Trends.* Figure 4(a) and Figure 4(b) show the results obtained for classification and consistency, respectively. Notice that both mobile devices follow the general trend obtained for the PC. For all the reasoners and devices, the average time for the small set is smaller than the time for the medium one, which in turn is smaller than for the large set. There are two situations that should require further explanation: first, *HermiT* was faster on A2 than on A1 for the classification of the small and medium sets of ontologies; and second, *Pellet* was as fast on A2 for the classification of the small set as for the medium one (while it was faster on the PC and A1). Analyzing the results per ontology of these ontology sets, we observed that there were some ontologies where the variance of the measured times for the three repetitions of each test was high, which led to these situations.

*Comparing Performance.* Table 8 shows the difference of performance between the PC and the two mobile devices in terms of number of times of the PC executions being faster than the Android ones. As in the DL ontology set, in general, the desktop computer outperformed the mobile devices for all the reasoners. The greatest differences are again achieved on the large and medium ontology sets. For example, for the classification of these sets using *HermiT*, PC is almost 70 times faster than A1, and almost 150 times faster than A2. In general, the differences between PC and the mobile devices are smaller for consistency checking. Notice that there is a situation where the Android devices were faster than the desktop computer: in the consistency checking with *ELK*, PC was 0.8 (from 0.26s to 0.21s) and 0.6 (from 0.26s to 0.15s) times "faster" than A1 and A2 devices, respectively (these values are coherent with the observed variance).

As it happens in the DL ontology set, the difference between the different reasoners on PC is smaller than on the mobile devices. Furthermore, the order of the reasoners according to their reasoning times is usually the same as in the desktop computer.




(a)




(b)

Figure 4: Average computing time for each ontology in the minimum set of OWL 2 EL ontologies processed by all the devices and reasoners.

## 7. Other Experiments

In this section we summarize some additional experiments measuring other interesting features of the mobile devices, namely the impact of the memory and the virtual machine.

### *7.1. Analyzing the Impact of Memory*

After comparing the results obtained for PC and Android, we wanted to check how the limitation of memory that Android imposes on applications and its management by the OS affects the results. Our goal was to check whether the processing time would be affected if the maximum memory for the application was limited.

| | | Classification | | | Consistency | | |
|---|---|---|---|---|---|---|---|
| | | S | M | L | S | M | L |
| ELK | A1 | 2 | 6 | 15 | 1 | 4 | 13 |
| | A2 | 1 | 7 | 23 | 1 | 4 | 21 |
| HermiT | A1 | 51 | 69 | 60 | 3 | 9 | 20 |
| | A2 | 14 | 47 | 148 | 3 | 9 | 33 |
| jcel | A1 | 3 | 12 | 33 | 7 | 20 | 54 |
| | A2 | 7 | 17 | 49 | 6 | 18 | 71 |
| JFact | A1 | 11 | 60 | 62 | 1 | 4 | 9 |
| | A2 | 12 | 64 | 92 | 1 | 5 | 20 |
| Pellet | A1 | 3 | 9 | 27 | 1 | 5 | 21 |
| | A2 | 31 | 16 | 36 | 1 | 5 | 37 |
| TrOWL | A1 | 8 | 27 | 20 | 8 | 28 | 24 |
| | A2 | 7 | 58 | 29 | 8 | 59 | 62 |

Table 8: Number of times (rounded to the closest integer) of the PC version being faster than the Android ones for the small (S), medium (M), and large (L) OWL 2 EL ontology set.

Firstly, we restricted the maximum memory for the desktop computer to 256 MB RAM (the maximum size that Android assigns to the applications in our test devices) and computed the classification of the DL and the EL ontology sets. We observed that there are significant differences on the number of finished tasks but not on the reasoning time. Figures 5 and 6 compare the number of finished classifications in PC, PC with the memory limitation (denoted PCmem), and A1 for the small (S), medium (M), and large (L) OWL 2 DL ontology set. In particular, we represent the differences between the number of finished tasks in the devices: PC vs. PCmem, PC vs. A1, and PCmem vs. A1.

We can see that the number of finished tasks over small ontologies is the same in all the cases. For medium ontologies, PC and PCmem only differ in one OWL 2 DL ontology, although A1 does not finish 14 OWL 2 DL and 5 OWL 2 EL ontologies. The only OWL 2 DL reasoner that does not finish a smaller number of tasks on A1 is TrOWL, which computes approximated reasoning in this profile. In large ontologies, the number of unfinished tasks on PCmem and A1 is significant: more than 10% in the EL ontology set and more than 50% in the DL ontology set. Overall, the number of finished tasks in PCmem and A1 is comparable: 586 vs 572 OWL 2 DL ontologies, and 1099 vs 1095 OWL 2 EL ontologies. Note that some tasks finished on A1 but not on PCmem.

After investigating the role of memory on the

desktop computer, we also limited the memory on the Android devices. To do that, we restricted the maximum memory heap for the application to the "standard memory heap" by setting the variable `android:largeHeap=''false''`. In particular, we restricted the memory to 96 MB. Since the experiments on Android devices are more costly, we restricted to the classification (the most challenging task) of the DL ontology set using Pellet and the Galaxy Nexus smartphone (A1, as it was the fastest device). For this experiment, we did not want to consider TrOWL (it only offers approximate reasoning in the DL ontology set) and JFact (because of the smaller number of finished classifications on A1); between HermiT and Pellet we chose the latter one to decrease the total computation time because, as shown in Figure 2 (a), it was usually faster.

The first aspect to highlight from the results is that, as it happened when limiting the memory on Android, limiting the memory decreased the number of finished tasks. The reasoner finished 143 tasks before and 135 after the memory limitation. These 8 tasks that could not be finished by the "limited version" include 2 medium and 6 large ontologies. Figure 7 shows the comparison of the time needed for every ontology that the reasoner was able to classify in the two tests. In the graph, we plot the processing times difference as a percentage of the time required by the non-limited version (y-axis) and the time needed by the non-limited version in seconds (x-axis). The first thing we can highlight is that there are some negative values which mean that the limited version was faster than the non-limited version. This can be explained because measuring time consumption of the same application on the same device can have a small variance due to the management of the running applications done by the OS. Moreover, for ontologies that can be classified quickly (under 5s), the difference is around 40%, and reaches even 60% in some cases. In these cases, the ontologies require around 1s to be classified, so the actual difference in seconds is around 0.5s. Regarding the values obtained, notice that the difference between the two versions for those ontologies that need 15s or more to be classified is less than 2%. This includes ontologies that needed 80s-250s, where the difference is less than 1s. Therefore, the main conclusion of this test is that limiting the memory on the devices do not significantly modifies the time consumption, but it does affect the number of accomplished tasks.

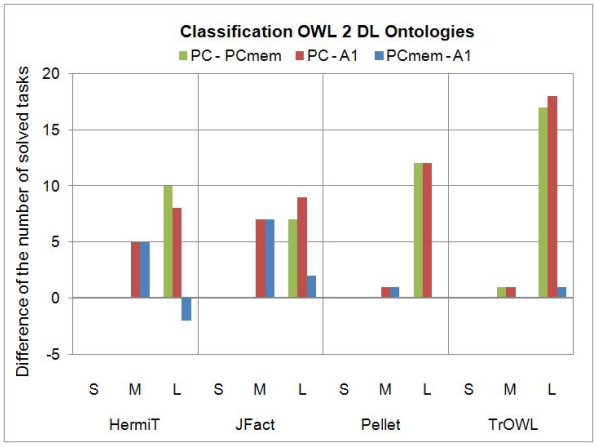



Figure 5: Comparison of the number of finished classifications of OWL 2 DL ontologies.

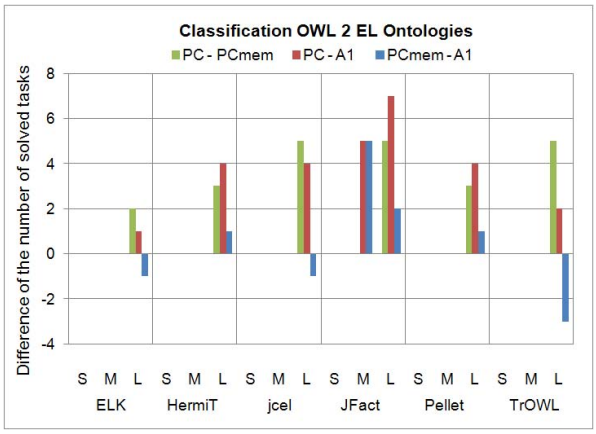



Figure 6: Comparison of the number of finished classifications of OWL 2 EL ontologies.

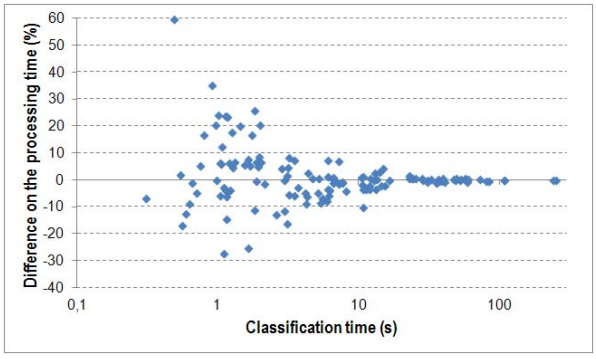



Figure 7: Comparison of the Pellet reasoner on Android with limited and "unlimited" memory.

### 7.2. Analyzing the Impact of the Virtual Machine

Very recently, as of November 2014, Google released the new Android 5.0 version that includes a new runtime environment called Android Runtime (ART) to replace the Dalvik virtual machine. While the previous virtual machine Dalvik uses just-in-time compilation every time an application is launched, ART uses a more sophisticated ahead-of-time compilation that can be performed just once during the installation of the application. This way, processor and battery usage are optimized. Therefore, the performance of mobile applications running on the new virtual machine is expected to increase. However, attending to historic information, it is expected that previous versions of the OS would continue maintaining their popularity. In fact, Android 4.X required a bit more than a year to reach 50% of the Android devices and, as of July 2015, previous versions were present on almost a 6% of the global devices[59].

We performed a final test with this new runtime environment to show the expected tendency in the future. We used a Google Nexus 5 smartphone (equipped with a 2.26 GHz quad-core processor and 2 GB of RAM) for this test. We computed the classification of 10 ontologies extracted from the DL ontology set which we used in our tests in [19] using Pellet. We ran this test twice, one with the current Android version (4.4) using the Dalvik virtual machine, and another one with the new Android version (5.0) using ART.

Figure 8 shows the comparison of the classification time on both virtual machines and for each of the 7 ontologies that finished the test (2 of them failed because of unsupported datatypes, and another one elapsed a time out). Notice that the new Android version and its virtual machine outperformed the previous one for all the ontologies using the same hardware. In fact, Pellet on the ART virtual machine was 2.5 times faster on average. As the tests were only performed once for each device we should take this number with a pinch of salt due to the variance of the times measured in our previous experiments. However, the improvement is similar with large, medium, and small ontologies so we can highlight that *reasoners on the new ART virtual machine could be around 2 times faster than in Dalvik*. Therefore, although reasoning on a desktop computer clearly continues outperforming reasoning on mobile devices, the processing times on mobile devices will go on decreasing as more powerful hardware and OS and software optimizations will be available, making reasoning on mobile devices even more feasible.

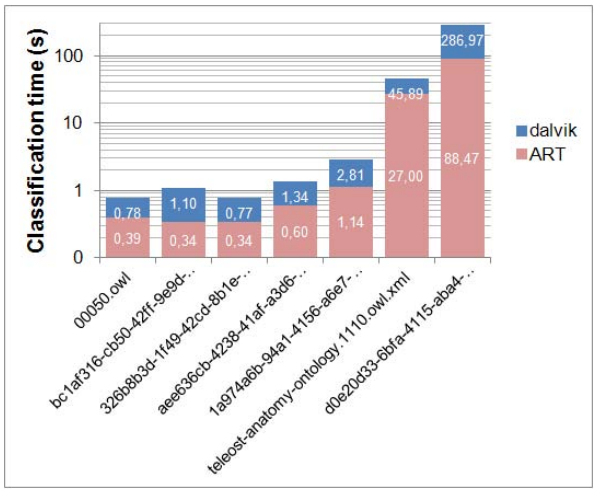



Figure 8: Comparison of the Pellet reasoner on the Dalvik and ART virtual machines.

## 8. Discussion

In this section, we present some key ideas extracted from our experience within this project. We believe that developers of existing and future semantic APIs and reasoners can benefit from taking into account these pieces of advice to enable mobile application (and specifically Android) developers to use their technologies. We also hope that developers of mobile applications that are considering using semantic technologies find these advices interesting when creating their applications.

Regarding the coding of semantic APIs and reasoners, we highlight the following points:

- Some of the reasoners we have studied and/or ported include many functionalities that might not be required for mobile application developers (e.g., graphical packages or remote access). We advocate for modular designs with a micro-kernel supporting only the core reasoning tasks, and with the rest of functionalities (e.g. parsers, query processing, etc.) included in separated modules. This would greatly help porting and using them, as the deployed version could be tailored and thus, some of the limitations of mobile devices could be avoided.

[59] `http://developer.android.com/about/dashboards/index.html`, last accessed 2015-07-04.

- Using native code to develop a reasoner can help the developer to avoid many of the limitations that the virtual machine imposes, achieving better performance. However, the difficulty of use of a reasoner within a mobile application/an Android project might be an unbearable barrier that could reduce the popularity of the reasoner. For this reason, there should be a simple, and if possible standard, mechanism to support its use from mobile applications. For that, we advocate providing always interfaces (such as OWL API or Jena) for mobile application developers.
- Beware of using some common standard Java libraries. Current Android/Dalvik versions do not perfectly align to a standard Java environment (e.g., all the graphical packages are not supported), and, even worse, there are libraries which are not completely included on current versions of Android and do not throw compilation or execution errors (e.g., JAXB and Xerces). Therefore, it is possible to import some reasoners on an Android project but the results of the reasoning are not the same as on desktop computers (as it happens in JFact 1.2.1).
- Keep in mind resource limitations. Regarding memory, in Android, we can reduce the memory imprint by using shared libraries. However, one might end up exceeding the maximal number of methods that can be invoked from a single dex file[60]. Thus, we advocate trying to use only those packages/classes needed and building a customized version of the library.
- As a final suggestion, using open-source licenses and distributing the code (as, for example, HermiT and Pellet) can help developers to find programmers willing to help in porting existing reasoners (as we have done) to other mobile OS, such as iOS.

Regarding the use and performance of semantic technologies on current Android devices:

- The performance of all the semantic reasoners we tested on Android is lower than on a PC. Indeed, the PC is from 1.5 to 150 times faster in our tests depending on the task and the ontology. Therefore, complicated tasks such as classification of large ontologies should be reduced to the minimum.
- The reasoners tested on Android behave similarly to the same version on a desktop computer: reasoners that are faster on desktop computer are generally faster on Android too.
- The variance of the reasoning time is higher on Android devices than on desktop computers. In the case of Android devices, the time variance is almost negligible for small ontologies, moderate for medium ontologies, and significant for larger ontologies. Ontologies in the OWL 2 DL ontology set usually produce higher variances than OWL 2 EL ontologies. Note that the main priority in Android is the responsiveness of the device, and, thus, its scheduling policies seem to penalize applications which require intensive use of resources (e.g., CPU time and memory). In general, DL ontologies require more computation time than EL ontologies, and thus, they are more prone to be affected by the variance introduced by the possible context changes.
- When developing a mobile semantic application, it is a good idea to separate the reasoning thread in an isolated process whenever possible, and not relying on Android to relaunch it if it gets killed (e.g., due to memory issues). If a task is killed for being too resource greedy, Android seems to penalize it and does not resume it immediately.
- To use the full potential of mobile devices, reasoners could compile the core of their code as Android native code and avoid this way the overhead of the Android virtual machine.
- Android uses `UTF-8` as encoding by default. Problems with the characters in the URIs can appear when working with ontologies that have been developed in an editor that stores them in any other encoding.

We also want to share some experiences with mobile application developers (researchers or not). First, with respect to the use of DL reasoners specifically designed for mobile devices, some authors of these reasoners argued that "current Semantic Web reasoners cannot be ported without a significant rewrite effort" [15]. After our work, this is no longer

[60] Although multidex mobile applications can be build since Android 4.0 (SDK 14), there are many documented problems with it, and we discourage its use, see `https://developer.android.com/tools/building/multidex.html` for details.

true, as we have made it possible to reuse existing semantic APIs and DL reasoners on Android and, in some cases, no rewriting was even needed. However, it is interesting to study if these reasoners outperform reused reasoners in mobile devices. In fact, most of the optimization techniques implemented by classical DL reasoners cannot easily be adopted in mobile systems, since they decrease running time but definitely increase the use of memory, which is limited in mobile devices. Thus, a study of the trade-off between expressivity and resource consumption for mobile devices could be useful.

In our experience, the use of semantic technologies on mobile devices allowed us to create smart mobile applications that do not require Internet connection and preserve the privacy of users by avoiding the use of cloud-based services. For example, SHERLOCK [4] uses JFact, while FaceBlock [11], Rafiki [9], and Triveni [57] use HermiT (see Section 1 for a short description of these applications). We only experienced problems when dealing with large ontologies (with hundreds of individuals) and complex SWRL rules. In those scenarios (large ontologies), the classification time exceeded the amount of time a user would be willing to wait for an answer.

## 9. Related Work

This section will be divided in two parts. Firstly, we will overview the previous work on supporting DL reasoners on mobile devices. Then, we will point to some relevant literature on empirical evaluations of DL reasoners on desktop computers.

### *9.1. Reusing and Evaluating DL Reasoners on Mobile Devices*

To the best of our knowledge, our work is the first effort to offer a systematic support and evaluation of the performance of existing DL reasoners on mobile devices. However, there are some previous works in the field that are worth mentioning.

Using ELK on Android devices has been recently investigated [61]. In particular, the authors implemented some minor changes to make the reasoner work on an Android smartphone and performed some experiments on a Google Nexus 4 Android 4.2 phone. In particular, the authors measured the classification time of 5 $\mathcal{EL}$ ontologies both in the Android device and in a desktop computer. The results show that reasoning times in the Android device are acceptable even if much slower (two orders of magnitude) than in the desktop version. We have also considered ELK in our experiments, measuring and comparing its reasoning times over more ontologies.

As previously mentioned, using Pellet on mobile (J2ME) devices has been investigated [49, 13]. The modification of Pellet reasoner included some new optimization techniques and was called *mTableau*. The authors did some experiments on a desktop computer, proving that the optimization is useful to reduce the response time in situations of limited memory. They also performed some experiments (4 consistency tests over 2 ontologies) in a PDA showing that the reasoning times are acceptable. However, their approach is more oriented to proving the usefulness of the optimizations rather than performing a comparison between the performance of the reasoner in mobile and desktop devices.

Finally, there are three recent works that consider the use of reasoners on mobile devices complementing our work. Regarding battery consumption, [58] analyzes the performance per watt of Jena, Pellet, and HermiT over two ontologies. The authors found a nearly linear relationship between energy consumption and processing time, and studied the effects of some smartphone features (WiFi, 3G, and 4G radios) on battery consumption. More recently, [59] studies the battery consumption on Android 5.x (API 21) of Pellet, Hermit, and Androjena, performing four different reasoning tasks over some datasets generated using the LUBM benchmark generator [60]. Their software-based approach could be used to extend our study on the performance of DL reasoners on current and future versions of Android. Finally, [62] presents a benchmark framework for mobile semantic reasoners allowing them to be deployed on different platforms such as Android or iOS. So far, the authors have only considered 4 reasoners (AndroJena, Nools, RDFQuery, and RDFStore-JS), since their work is more focused on generability and extensibility, making it easier to add new mobile platforms and reasoners.

The main novelties of this paper with respect to our previous work [18, 19] are a more detailed description of the porting process, the study of more semantic reasoners (MORe, TrOWL, and TReasoner), reasoning tasks (consistency), and ontologies (the whole set of ontologies used in ORE 2013), new experiments (with a new evaluation of the roles of memory and virtual machine in Android devices),

and some recommendations for application and reasoner developers.

### *9.2. Evaluating DL Reasoners on Desktop Computers*

The developers of some reasoners have performed evaluations of their systems with the main objective of showing that their new tools outperformed existing ones. There are also several (more or less) independent experimental comparisons in the literature that we will overview here.

In [63], FaCT++, Pellet 1.1.0, and Racer 1.8.0 reasoners were compared for the classification of 135 OWL ontologies, with FaCT++ being slightly preferable (faster and more robust). Then, in [64], FaCT++ 1.1.3, Pellet 1.3, and RacerPro 1.8.1, were compared along with KAON2 for the classification of 172 ontologies, without a clear winner due to the considerable difference of performance across ontologies.

HermiT, KAON2, Pellet, RacerPro, Sesame, and SwiftOWLIM reasoners were compared for classification and conjunctive query answering over 3303 ontologies in [65]. The authors concluded that SwiftOWLIM may be preferable in low expressive languages, RacerPro can be recommended in expressive ontologies with small ABoxes, and KAON2 is the best alternative in the other cases.

The reasoners CB, CEL, DB, FaCT++, and HermiT were evaluated for the classification of 4 $\mathcal{ELH}$ ontologies [36]. CB is the best option, although the results are focused on evaluating the new technique of computing classification using an SQL system.

Dentler et al. [66] evaluated 8 reasoners (CB build 6, CEL 0.4.0, FaCT++ 1.5.0, HermiT 1.3.0, Pellet 2.2.2, RacerPro 2.0 preview, Snorocket 1.3.2, and TrOWL 0.5.1) over 3 OWL 2 EL ontologies for 4 reasoning tasks (classification, concept satisfiability, TBox consistency, and subsumption). As usual, there is not a clear winner. Besides, the authors also performed a very detailed comparison of other features of the reasoners.

Another experiment measures the classification time of 358 real-world ontologies for 4 reasoners (FaCT++, HermiT, Pellet, and TrOWL) [67]. The best reasoner depends on the criteria: Fact++ has the lowest median, HermiT has the lowest mean, TrOWL has the lowest number of errors, and Pellet has the lowest number of errors among the complete reasoners.

Since DLs usually have a good performance in practice but high worst-case complexities, [68] investigates how often reasoning with existing ontologies requires an unreasonable time. The authors consider 4 reasoners (FaCT++ 1.6.1, HermiT 1.3.6, JFact 1.0, and Pellet 2.3.0) and 1071 ontologies, showing that most of the times there is some reasoner giving a quick response time, with Pellet being the most robust one. Hence, most of the existing ontologies on the Web are not inherently intractable but just hard for some particular DL reasoners.

Finally, it is worth to mention that the OWL Reasoner Evaluation Workshop (ORE) series organize DL reasoner competitions. The 2012 competition[61] considered 143 ontologies and 5 reasoning tasks (classification, consistency, concept satisfiability, entailment, and instance retrieval) for 4 reasoners (FaCT++, HermiT, jcel, and WSReasoner). The 2013 competition [52] considered 204 ontologies classified in 3 profiles (OWL 2 DL, OWL 2 EL, and OWL 2 RL) and 3 reasoning tasks (classification, consistency, and concept satisfiability) for 14 reasoners (BaseVISor, Chainsaw, ELepHant, ELK, FaCT++, HermiT, jcel, JFact, Konclude, MORe, SnoRocket, Treasoner, TrOWL, and WSClassifier). The competition organizers gave priority to robustness of the systems rather than the reasoning times alone. The latest editions held at 2014 [53] and 2015 [54] considered more than 16500 unique ontologies divided in 2 profiles (OWL 2 DL, and OWL 2 EL), and 3 reasoning tasks (classification, consistency checking, and realisation). The number of ontologies used out from the ontology set depended on the profile and the reasoning task, ranging from 200 ontologies used for realization in DL profile to 300 ontologies used for classification in EL profile. The participants in the 2014 edition were Chainsaw, ELepHant, ELK, FaCT++, HermiT, jcel, JFact, Konclude, MORe, Treasoner, and TrOWL (11 reasoners), and the participants in the 2015 edition were Chainsaw, ELephant, ELK, FaCT++, HermiT, jcel, JFact, Konclude, MORe, PAGOdA, Pellet, Racer, and TrOWL (13 reasoners). In all the competitions, the best reasoner depends on the reasoning task and the expressivity of the ontology.

## 10. Conclusions and Future Work

The emergence of mobile computing in our daily lives requires considering new applications where

[61] http://www.cs.ox.ac.uk/isg/conferences/ORE2012/evaluation/index.html

semantic technologies will be useful. However, some efforts are needed to enable developers to use ontologies and ontology reasoning in their mobile applications.

In this paper, we have shown that using semantic technologies on current mobile devices is feasible. Focusing on Android-based devices, we have been able to use most of the available semantic reasoners. In particular, we were able to use CB, ELK, HermiT, jcel, JFact, MORe, Pellet, TReasoner, and TrOWL. However, our ongoing work shows that using semantic reasoners on mobile devices is not trivial. In general, some manual work is needed due to the presence of unsupported Java libraries and classes in Dalvik, making some rewriting efforts necessary. We have detailed the changes needed to make some reasoners work, hoping that this will make porting future versions easier. Unfortunately, it turned out that newer versions of some reasoners (HermiT and Pellet) are not easier to port than older ones, and we have also experienced some new errors with JFact versions starting from 1.0.

In order to test the reasoners on Android devices, we have designed some experiments with standard ontology sets used in the OWL Reasoner Evaluation 2013 workshop. We considered using the ontology set used in the 2014 workshop, but the number of ontologies (16555) is huge. First, we have tested that the ported versions on Android obtain the same results than the original versions on PCs. Then, we have tested the performance of the reasoners in terms of number of tasks finished and time consumed per task in two mobile devices, a smartphone and a tablet. Finally, we have tested the role of the amount of available memory and the virtual machine used on Android. The complete results of our experiments can be found on the webpage [12], together with a detailed description of all the changes needed to port the semantic reasoners and, if the licenses make it possible, download links.

From a practical point of view, the main limitation that reasoners will face on current smartphones/tablets concerns memory usage and processing time (and hence, battery consumption). Our experiments show that reasoners running on a PC are 1.5 to 150 times faster than on Android devices. Also, the number of out of memory errors increase on Android devices compared with PCs, usually in the OWL 2 DL profile and in larger ontologies. We also noticed important differences in the performance of the three analyzed Android devices, showing that, although mobile devices are far from being desktop computers, they are increasing their capabilities quickly as needed by challenging tasks, such as semantic reasoning. We have recently done some tests with more modern devices that confirm this trend. In addition, we have shown some results with the future Android runtime, ART, that show that the same tasks on the same devices can be executed around 2 times faster.

We would like to finish this summary of the conclusions of the paper with a statistical curiosity. We estimate that the complete empirical experimentation reported in this paper required a total computing time of more than 1000 hours (which is more than 41 days only for the computation of the different tests).

As future work we would like to port and evaluate more semantic reasoners. We are especially interested in reasoners for OWL 2 QL and OWL 2 RL profiles, as we have not been able so far to port any reasoner specifically designed for such languages, and in the evaluation of CB over Horn ontologies. Moreover, to extend the insight on the capabilities of mobile devices, we are interested on evaluating other reasoning tasks such as query answering or realization, as they involve ABox reasoning and are interesting in many mobile scenarios; and on studying the impact of using the reasoners through their own APIs instead of the OWL API. Finally, we would like to extend our study to include the mobile device's battery in order to show the feasibility of using DL reasoners in real applications. We plan to do so by measuring the energy consumption (similarly as in [58] and [59]), and by analysing the influence of the battery level in the performance of the different reasoners in disconnected scenarios (devices not plugged to any power source).

## Acknowledgments

This research work has been supported by the CICYT projects TIN2010-21387-C02-02, TIN2013-46238-C4-4-R, and DGA-FSE. We would like to thank Guillermo Esteban, for his help with the porting of the reasoners to Android, and the anonymous reviewers for their valuable comments on an earlier version of this paper.

[1] T. R. Gruber, A translation approach to portable ontology specifications, Knowledge Acquisition 5 (2) (1993) 199–220.

[2] A. Sinner, T. Kleemann, KRHyper In your pocket, in: 20th Intl. Conf. on Automated Deduction (CADE-20), Vol. 3632, LNCS, Springer, 2005, pp. 45–458.

[3] T. Kleemann, Towards mobile reasoning, in: 2006 Intl. Workshop on Description Logics (DL 2006), 2006.
[4] R. Yus, E. Mena, S. Ilarri, A. Illarramendi, SHERLOCK: Semantic management of location-based services in wireless environments, Pervasive and Mobile Computing 15 (2014) 87–99.
[5] C. Becker, C. Bizer, Exploring the geospatial Semantic Web with DBpedia Mobile, Journal of Web Semantics.
[6] M. L. Wilson, A. Russell, D. A. Smith, A. Owens, M. M. C. Schraefel, mSpace Mobile: A mobile application for the Semantic Web, in: 2nd Intl. Workshop on Interaction Design and the Semantic Web, 2005.
[7] N. Ambroise, S. Boussonnie, A. Eckmann, A smartphone application for chronic disease self-management, in: 2013 Mobile and Information Technologies in Medicine and Health Conference (MobileMED 2013), 2013.
[8] S. R. Abidi, S. S. R. Abidi, A. Abusharekh, A semantic web based mobile framework for designing personalized patient self-management interventions, in: 2013 Mobile and Information Technologies in Medicine and Health Conference (MobileMED 2013), 2013.
[9] P. Pappachan, R. Yus, A. Joshi, T. Finin, Rafiki: A semantic and collaborative approach to community health-care in underserved areas, in: Proc. of the 10th IEEE Intl. Conf. on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2014), 2014.
[10] D. Ghosh, A. Joshi, T. Finin, P. Jagtap, Privacy control in smart phones using semantically rich reasoning and context modeling, in: 2012 IEEE Symposium on Security and Privacy Workshops (SPW 2012), 2012, pp. 82–85.
[11] R. Yus, P. Pappachan, P. K. Das, E. Mena, A. Joshi, T. Finin, FaceBlock: Privacy-aware pictures for Google Glass, in: 12th Annual Intl. Conf. on Mobile Systems, Applications, and Services (MobiSys 2014), 2014, pp. 366–366.
[12] Android goes semantic!, `http://sid.cps.unizar.es/AndroidSemantic`.
[13] L. Steller, S. Krishnaswamy, M. M. Gaber, Enabling scalable semantic reasoning for mobile services, International Journal on Semantic Web and Information Systems 5 (2) (2009) 91–116.
[14] B. Motik, I. Horrocks, S. M. Kim, Delta-reasoner: A Semantic Web reasoner for an intelligent mobile platform, in: 21st World Wide Web Conference (WWW 2012), Companion Volume, 2012, pp. 63–72.
[15] M. Ruta, F. Scioscia, E. D. Sciascio, F. Gramegna, G. Loseto, Mini-ME: the mini matchmaking engine, in: 1st Intl. Workshop on OWL Reasoner Evaluation (ORE 2012), Vol. 858, CEUR-WS, 2012.
[16] F. Müller, M. Hanselmann, T. Liebig, O. Noppens, A tableaux-based mobile DL reasoner - An experience report, in: 19th Intl. Workshop on Description Logics (DL 2006), Vol. 189, CEUR-WS, 2006.
[17] E. Burnette, Hello, Android: Introducing Google's Mobile Development Platform, The Pragmatic Programmers, LLC., 2010.
[18] R. Yus, C. Bobed, G. Esteban, F. Bobillo, E. Mena, Android goes semantic: DL reasoners on smartphones, in: 2nd Intl. Workshop on OWL Reasoner Evaluation (ORE 2013), Vol. 1015, CEUR-WS, 2013, pp. 46–52.
[19] C. Bobed, F. Bobillo, R. Yus, G. Esteban, E. Mena, Android went semantic: Time for evaluation, in: 3rd Intl. Workshop on OWL Reasoner Evaluation (ORE 2014), Vol. 1207, CEUR-WS, 2014, pp. 23–29.
[20] B. McBride, Jena: A Semantic Web toolkit, IEEE Internet Computing 6 (6) (2002) 55–59.
[21] M. Horridge, S. Bechhofer, The OWL API: A Java API for OWL ontologies, Semantic Web Journal 2 (1) (2011) 11–21.
[22] Y. Kazakov, Consequence-driven reasoning for Horn $\mathcal{SHIQ}$ ontologies, in: 21st Intl. Joint Conf. on Artificial intelligence (IJCAI 2009), 2009, pp. 2040–2045.
[23] Y. Kazakov, M. Krötzsch, F. Simančík, The incredible ELK, Journal of Automated Reasoning 53 (2014) 1–61.
[24] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, Z. Wang, HermiT: An OWL 2 reasoner, Journal of Automated Reasoning 53 (3) (2014) 245–269.
[25] J. Mendez, jcel: A modular rule-based reasoner, in: 1st Intl. Workshop on OWL Reasoner Evaluation (ORE 2012), Vol. 858, CEUR-WS, 2012.
[26] F. Baader, C. Lutz, B. Suntisrivaraporn, CEL - A polynomial-time reasoner for life science ontologies, in: 3rd Intl. Joint Conf. on Automated Reasoning (IJCAR 2006), Vol. 4130, LNAI, Springer, 2006, pp. 287–291.
[27] D. Tsarkov, I. Horrocks, FaCT++ description logic reasoner: system description, in: 3rd Intl. Joint Conf. on Automated Reasoning (IJCAR 2006), 2006.
[28] I. Horrocks, Using an expressive description logic: FaCT or fiction?, in: Proc. of the 6th Intl. Conf. on the Principles of Knowledge Representation and Reasoning (KR 1998), Morgan Kaufmann Publishers, 1998, pp. 636–647.
[29] A. Armas-Romero, B. Cuenca-Grau, I. Horrocks, E. Jiménez-Ruiz, MORe: a modular owl reasoner for ontology classification, in: 2nd Intl. Workshop on OWL Reasoner Evaluation (ORE 2013), Vol. 1015, CEUR-WS, 2013, pp. 61–67.
[30] E. Sirin, B. Parsia, B. Cuenca-Grau, A. Kalyanpur, Y. Katz, Pellet: A practical OWL-DL reasoner, Journal of Web Semantics 5 (2) (2007) 51–53.
[31] E. Thomas, J. Z. Pan, Y. Ren, TrOWL: Tractable OWL 2 reasoning infrastructure, in: 7th Extended Semantic Web Conference (ESWC 2010), 2010.
[32] A. V. Grigorev, A. G. Ivashko, TReasoner: System description, in: Proc. of the 2nd Intl. Workshop on OWL Reasoner Evaluation (ORE 2013), Vol. 1015, CEUR-WS, 2013, pp. 26–31.
[33] C. J. Matheus, K. Baclawski, M. M. Kokar, Basevisor: A triples-based inference engine outfitted to process RuleML and r-entailment rules, in: Proc. of the 2nd Intl. Conf. on Rules and Rule Markup Languages for the Semantic Web (RuleML 2006), 2006, pp. 67–74.
[34] D. Tsarkov, I. Palmisano, Chainsaw: a metareasoner for large ontologies, in: Proc. of the 1st Intl. Workshop on OWL Reasoner Evaluation (ORE 2012), Vol. 858, CEUR-WS, 2012.
[35] F. Simanck, Y. Kazakov, I. Horrocks, Consequence-based reasoning beyond Horn ontologies, in: 22nd Intl. Joint Conf. on Artificial Intelligence (IJCAI 2011), 2011, pp. 1093–1098.
[36] V. Delaitre, Y. Kazakov, Classifying $\mathcal{ELH}$ ontologies in SQL databases, in: 6th Intl. Workshop on OWL: Experiences and Directions (OWLED 2009), Vol. 529, CEUR-WS, 2009.
[37] B. Sertkaya, The ELepHant reasoner system description, in: 2nd Intl. Workshop on OWL Reasoner Evaluation (ORE 2013), Vol. 1015, CEUR-WS, 2013, pp.

87–93.

[38] F. Bobillo, U. Straccia, fuzzyDL: An expressive fuzzy Description Logic reasoner, in: 17th IEEE Intl. Conf. on Fuzzy Systems (FUZZ-IEEE 2008), 2008, pp. 923–930.

[39] B. Motik, R. Studer, KAON2–a scalable reasoning tool for the Semantic Web, in: 2nd European Semantic Web Conference (ESWC 2005), 2005.

[40] A. Steigmiller, T. Liebig, B. Glimm, Konclude: System description, Journal of Web Semantics 27–28 (2014) 78–85.

[41] B. Bishop, A. Kiryakov, D. Ognyanoff, Z. Tashev, R. Velkov, OWLIM: A family of scalable semantic repositories, Semantic Web Journal 2 (2011) 33–42.

[42] V. Haarslev, K. Hidde, R. Möller, M. Wessel, The RacerPro knowledge representation and reasoning system, Semantic Web Journal 3 (2012) 267–277.

[43] J. Dolby, A. Fokoue, A. Kalyanpur, E. Schonberg, K. Srinivas, Scalable highly expressive reasoner (SHER), Journal of Web Semantics 7 (4) (2009) 357–361.

[44] J. Lu, L. Ma, L. Zhang, J.-S. Brunner, C. Wang, Y. Pan, Y. Yu, SOR: A practical system for ontology storage, reasoning and search, in: Proc. of the 33rd Intl. Conf. on Very Large Data Bases (VLDB 2007), ACM, 2007, pp. 1402–1405.

[45] M. Lawley, C. Bousquet, Fast classification in Protégé: Snorocket as an OWL 2 EL reasoner, in: Australasian Ontology Workshop 2010 (AOW 2010), 2010, pp. 45–50.

[46] W. Song, B. Spencer, W. Du, A transformation approach for classifying $\mathcal{ALCHI}(\mathbf{D})$ ontologies with a consequence-based $\mathcal{ALCH}$ reasoner, in: 2nd Intl. Workshop on OWL Reasoner Evaluation (ORE 2013), Vol. 1015, CEUR-WS, 2013, pp. 39–45.

[47] W. Song, B. Spencer, W. Du, WSReasoner: a prototype hybrid reasoner for $\mathcal{ALCHOI}$ ontology classification using a weakening and strengthening approach, in: 1st Intl. Workshop on OWL Reasoner Evaluation (ORE 2012), Vol. 858, CEUR-WS, 2012.

[48] C. Wernhard, System description: KRHyper, Tech. rep., Fachberichte Informatik 142003, Universität Koblenz-Landau (2003).

[49] L. Steller, S. Krishnaswamy, Pervasive service discovery: mTableaux mobile reasoning, in: I-SEMANTICS 2008, Springer, 2008.

[50] M. Ruta, E. D. Sciascio, F. Scioscia, Concept abduction and contraction in semantic-based P2P environments, Web Intelligence and Agent Systems 9 (2011) 179–207.

[51] H.-S. Oh, B.-J. Kim, H.-K. Choi, S.-M. Moon, Evaluation of Android Dalvik virtual machine, in: 10th Intl. Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2012), 2012.

[52] R. S. Gonçalves, S. Bail, E. Jiménez-Ruiz, N. Matentzoglu, B. Parsia, B. Glimm, Y. Kazakov, OWL Reasoner Evaluation (ORE) workshop 2013 results: Short report, in: 2nd Intl. Workshop on OWL Reasoner Evaluation (ORE 2013), Vol. 1015, CEUR-WS, 2013, pp. 1–18.

[53] S. Bail, B. Glimm, E. Jiménez-Ruiz, N. Matentzoglu, B. Parsia, A. Steigmiller, Summary ORE 2014 competition, in: 3rd Intl. Workshop on OWL Reasoner Evaluation (ORE 2014), Vol. 1207, CEUR-WS, 2014, pp. IV–VII.

[54] S. Dumontier, B. Glimm, R. S. Gonçalves, M. Horridge, E. Jiménez-Ruiz, N. Matentzoglu, B. Parsia, G. Stamou, G. Stoilos, Summary ORE 2015 competition, in: 4th Intl. Workshop on OWL Reasoner Evaluation (ORE 2015), Vol. 1387, CEUR-WS, 2015, pp. IV–VI.

[55] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, S. Hellmann, DBpedia - A crystallization point for the Web of Data, Journal of Web Semantics 7 (3) (2009) 154–165.

[56] F. Baader, D. Calvanese, D. McGuinnes, D. Nardi, P. Patel-Schneider, The Description Logic handbook. Theory, implementation and applications, Cambridge University Press, 2003.

[57] R. Yus, P. Pappachan, P. K. Das, T. Finin, A. Joshi, E. Mena, Semantics for privacy and shared context, in: 2nd Intl. Workshop on Society, Privacy and the Semantic Web - Policy and Technology (PrivOn 2014), 2014.

[58] E. Patton, D. McGuinness, A power consumption benchmark for reasoners on mobile devices, in: 13th Intl. Semantic Web Conference (ISWC 2014), Part I, Vol. 8796, LNCS, Springer, 2014, pp. 409–424.

[59] E. Valincius, H. Nguyen, J. Z. Pan, A power consumption benchmark framework for ontology reasoning on Android devices, in: 4th Intl. Workshop on OWL Reasoner Evaluation (ORE 2015), Vol. 1387, CEUR-WS, 2015, pp. 80–86.

[60] Y. Guo, Z. Pan, J. Heflin, LUBM: A benchmark for OWL knowledge base systems, Journal of Web Semantics 3 (2-3) (2005) 158–182.

[61] Y. Kazakov, P. Klinov, Experimenting with ELK reasoner on Android, in: Proc. of the 2nd Intl. Workshop on OWL Reasoner Evaluation (ORE 2013), Vol. 1015, CEUR-WS, 2013, pp. 68–74.

[62] W. V. Woensel, N. A. Haider, A. Ahmad, S. S. R. Abidi, A cross-platform benchmark framework for mobile semantic web reasoning engines, in: Proc. of the 13th Intl. Semantic Web Conference (ISWC 2014), Part I, Vol. 8796, LNCS, Springer, 2014, pp. 389–408.

[63] Z. Pan, Benchmarking DL reasoners using realistic ontologies, in: Proc. of the workshop on OWL: Experiences and Directions (OWLED 2005), Vol. 188, CEUR-WS, 2005.

[64] T. Gardiner, D. Tsarkov, I. Horrocks, Framework for an automated comparison of description logic reasoners, in: 5th Intl. Semantic Web Conference (ISWC 2006), Vol. 4273, LNCS, Springer, 2006, pp. 654–667.

[65] J. Bock, P. Haase, Q. Ji, R. Volz, Benchmarking OWL reasoners, in: Proc. of the Intl. Workshop on Advancing Reasoning on the Web: Scalability and Commonsense (ARea 2008), 2008.

[66] K. Dentler, R. Cornet, A. ten Teije, N. de Keizer, Comparison of reasoners for large ontologies in the OWL 2 EL profile, Semantic Web Journal 2 (2) (2011) 71–87.

[67] Y. B. Kang, Y. F. Li, S. Krishnaswamy, A rigorous characterization of classification performance - A tale of four reasoners, in: 1st Intl. Workshop on OWL Reasoner Evaluation (ORE 2012), Vol. 858, CEUR-WS, 2012.

[68] R. S. Gonçalves, N. Matentzoglu, B. Parsia, U. Sattler, The empirical robustness of description logic classification, in: 26th Intl. Workshop on Description Logics (DL 2013), Vol. 1014, CEUR-WS, 2013, pp. 197–208.