A Lightweight Matchmaking Engine for the Semantic Web of Things

F. Gramegna, S. Ieva, G. Loseto, M. Ruta, F. Scioscia, and E. Di Sciascio

Politecnico di Bari, via E. Orabona 4, I-70125 Bari, Italy
E-mail: {gramegna, ieva, loseto}@deemail.poliba.it, {m.ruta, f.scioscia, disciascio}@poliba.it

Abstract. The Semantic Web of Things (SWoT) blends the Semantic Web and the Internet of Things visions. Due to architectural and performance issues, it is currently impractical to use available reasoners for processing semantic-based information in pervasive computing scenarios. This paper presents a prototypical mobile matchmaker for the SWoT, supporting Semantic Web technologies and implementing both standard and non-standard inference tasks for moderately expressive knowledge bases. Architectural and functional features are presented and an experimental performance evaluation is provided both on PCs and smartphones.

1 Introduction

The Semantic Web initiative envisions software agents to share, reuse and combine data available in the World Wide Web, by leveraging machine-understandable annotation languages such as RDF[1] and OWL[2], grounded on Description Logics (DLs). The Internet of Things [6], on the other hand, draws paradigms and approaches aiming to give intelligence to objects and locations by assigning information fragments to heterogeneous micro-devices and letting them to permeate the environment.

The Semantic Web of Things (SWoT) emerges today as an articulate effort to join together Semantic Web and the Internet of Things. Its goal is to embed semantically rich and easily accessible metadata into the physical world, by enabling storage and retrieval of annotations from tiny smart objects. Such a vision requires to enhance Semantic Web schemes and protocols for information dissemination and discovery because pervasive computing devices (*e.g.*, sensors, electro-mechanical components, smartphones, tablets), albeit increasingly effective and powerful, are still affected by hardware and software limitations. Particularly, reasoning and query answering devoted to object discovery are critical issues. A trivial one-to-one porting of OWL-based reasoners currently adopted

[1] Resource Description Framework, W3C Recommendation 10 February 2004, http://www.w3.org/TR/rdf-primer/

[2] OWL 2 Web Ontology Language, W3C Recommendation 11 December 2012, http://www.w3.org/TR/owl2-overview/

in the Semantic Web to handheld devices is affected by significant architectural and performance issues. Hence it is preferable to design specific mobile matchmakers, which –among other– have several peculiarities w.r.t. remote Web-based systems: (i) Internet connection not needed and no communication latency; (ii) competitive response times due to optimized inference procedures; (iii) greater reliability since remote systems can incur in bottleneck and single point of failure problems.

This paper presents a lightweight prototypical reasoning engine for moderately expressive DLs, created to support *semantic-based matchmaking* [11], [5] in pervasive computing scenarios oriented to the Web of Things (wireless semantic sensor and actor networks [26], semantic-enhanced mobile navigation [30] and driving assistance [29], domotics [25]). It complies with standard Semantic Web technologies through the OWL API [18] and implements both standard reasoning tasks for Knowledge Base (KB) management (subsumption, classification, satisfiability) and non-standard inference services for semantic-based resource discovery and ranking (abduction, contraction and covering [23, 12]). The tool is implemented in Java, adopting Android as target computing platform. Architectural details and algorithmic features are reported along with an early experimental campaign aimed to evidence performances obtained letting run the micro-reasoner on a PC and on a smartphone.

The remaining of the paper is organized as follows. Details about implemented reasoning algorithms and supported logic languages are given in Section 2, while Section 3 describes the software architecture. A case study is reported in Section 4, followed by an experimental evaluation of the tool in Section 5. Section 6 reports on related work, providing perspective and motivation for the proposal. Finally conclusion and future work in Section 7 close the paper.

2 Reasoning Services

In DL-based reasoning, an ontology $\mathcal{T}$ (a.k.a. Terminological Box or TBox) is composed by a set of axioms in the form: $A \sqsubseteq D$ or $A \equiv D$ where A and D are concept expressions. Particularly, a *simple-TBox* $\mathcal{T}$ complies with the following constraints: (i) $\mathcal{T}$ is acyclic; (ii) A is always an atomic concept; (iii) if A appears in the left hand side (lhs) of an equivalence axiom, then it cannot appear also in the lhs of any concept inclusion axiom. $\mathcal{ALN}$ (Attributive Language with unqualified Number restrictions) is a DL having polynomial computational complexity for standard and non-standard inferences in simple-TBoxes where the taxonomy depth is bounded by the logarithm of the axioms (see [14] for further explanations).

The matchmaker presented here works on KBs grounded on the simple-TBox hypothesis and implements a structural variant of standard and non-standard reasoning algorithms in $\mathcal{ALN}$. Given a KB $\mathcal{T}$, particularly the *unfolding* and the *Conjunctive Normal Form* (CNF) *normalization* is used to pre-process "una tantum" the knowledge base for applying further inferences. The **unfolding** procedure recursively expands references to axioms in $\mathcal{T}$ within the concept ex-

pression itself. In this way, $\mathcal{T}$ is not needed any more when executing subsequent inferences. The **normalization** transforms the unfolded concept expression in CNF by applying a set of pre-defined substitutions. Any concept expression C can be reduced in CNF as $C \equiv C_{CN} \sqcap C_{LT} \sqcap C_{GT} \sqcap C_{\forall}$:
- C_{CN} is the conjunction of (possibly negated) atomic concept names;
- C_{LT} (respectively C_{GT}) is the conjunction of $\leq$ (resp. $\geq$) number restrictions (no more than one per role);
- $C_{\forall}$ is the conjunction of universal quantifiers (no more than one per role; fillers are recursively in CNF).
Normalization preserves semantic equivalence w.r.t. models induced by the TBox; furthermore, CNF is unique (up to commutativity of conjunction operator) [14]. The normal form of an unsatisfiable concept is simply $\bot$. The following standard reasoning services on (unfolded and normalized) concept expressions are currently supported:
- **Subsumption test**. The classic structural subsumption algorithm is exploited, reducing the procedure to a set containment test [4].
- **Concept Satisfiability** (a.k.a. consistency). Due to CNF properties, satisfiability check is trivially performed during normalization.

The implemented reasoning services over ontologies are:
- **Ontology Coherence/Satisfiability**: since the proposed reasoner does not currently process the ABox, it performs an ontology *coherence* check rather than a satisfiability check (difference is discussed *e.g.*, in [19]).
- **Classification**: it computes the overall concept taxonomy induced by the subsumption relation, from $\top$ to $\bot$ concept. In order to reduce the subsumption tests, the following optimizations introduced in [1] have been taken into account: *enhanced traversal top search* and *bottom search*, exploitation of *told subsumers*.

Three non-standard inference services are also available, allowing to (i) enable a logic-based relevance ranking of a set of available resources w.r.t. a specific query and (ii) provide explanation of outcomes beyond the trivial "yes/no" answer of satisfiability and subsumption tests:
- **Concept Abduction** [23]: given a request D and a supplied resource S, if D and S are compatible but S does not imply D, Abduction allows to determine what should be hypothesized in S in order to completely satisfy D. The solution H (for *Hypothesis*) to Abduction represents "why" the subsumption relation $\mathcal{T} \models S \sqsubseteq D$ does not hold. H can be interpreted as *what is requested in D and not specified in S*.
- **Concept Contraction** [23]: if D and S are not compatible with each other, Contraction determines which part G (for *Give up*) of D is conflicting with S. If one retracts G, a concept K (for *Keep*) is obtained, representing a contracted version of the original request, such that $K \sqcap S$ is now satisfiable w.r.t. $\mathcal{T}$. The solution G to Contraction represents "why" $D \sqcap S$ are not compatible.
- **Concept Covering** [12]: given a request D and a set of resources $S = \{S_1, S_2, ..., S_k\}$, where D and S_1, S_2, ..., S_k are satisfiable in $\mathcal{T}$, the Concept Covering Problem (CCoP) aims to find a pair $\langle S_c, H \rangle$ where S_c includes concepts in S

(partially) covering D w.r.t. $\mathcal{T}$ and H is the (possible) part of D not covered by concepts in S_c.

3 System Architecture

The matchmaker prototype is compliant with the Android Platform version 2.1 (API level 7), therefore it runs on all devices based on Android 2.1 or later. It can be run either through the *Android Service* by Android applications, or as a library by directly calling the public methods of the *OwlReasoner* and *MicroReasoner* components shown in what follows. The system supports OWL 2 ontology language, in all syntaxes accepted by the OWL API parser.

The proposed architecture is sketched as UML diagram in Figure 1. Main components are: (i) **High Level Data Structures**: in-memory data structures for concept manipulation and reasoning; they refer to reasoning tasks on single concept expressions (concept satisfiability, subsumption, abduction, contraction, covering); (ii) **KB Wrapper**: implements KB management functions (creation of internal data structures, normalization, unfolding) and basic reasoning tasks on ontologies (classification and consistency check); (iii) **MicroReasoner**: interface for non-standard reasoning tasks (concept abduction, contraction, covering); (iv) **OwlReasoner**: OWL API [18] interface implementation which offers fundamental on-KB operations (load, parse) and standard reasoning tasks (subsumption, classification, satisfiability); it is backed by the OWL API open source library; (v) **Android Service**: implements a background daemon as interface toward applications willing to use the matchmaking engine. The service starts only when a client is connected and requires to execute an inference task; the server is multithread. A request has to be composed by a string command specifying the selected reasoning task and possible parameters; the reply will be formatted as XML text.

The UML diagram in Figure 2 depicts classes in the *High Level Data Structures* package (mentioned above) and their relationships. Standard Java Collection Framework classes are used as low-level data structures. (i) **Item**: each concept in the ontology is an instance of this class. Attributes are the name and the corresponding concept expression. When parsing an ontology, the *KB Wrapper* component builds a Java *HashMap* object containing all concepts in the TBox as *String-Item* pairs. Each concept is unfolded, normalized and stored in the

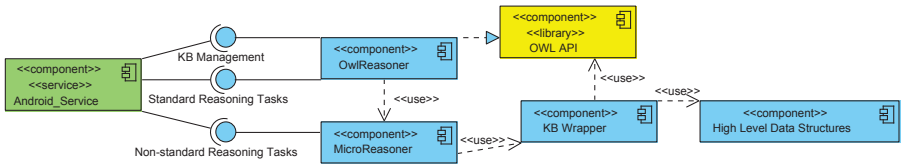


Fig. 1. Component UML diagram

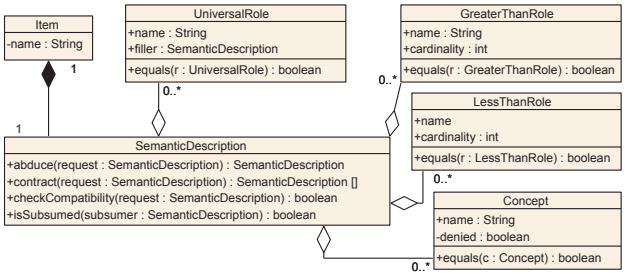


Fig. 2. Class diagram of *High Level Data Structures* package

HashMap with its IRI as key and *Item* instance as value. (ii) **SemanticDescription**: models a concept expression in CNF as aggregation of $C_{CN}, C_{GT}, C_{LT}, C_{\forall}$ components, each one stored in a different Java *ArrayList*. Methods implement inference services: `abduce` returns the hypothesis H expression; `contract` returns a two-element array with G and K expressions; `checkCompatibility` checks consistency of the conjunction between the object *SemanticDescription* and the one passed as input parameter; similarly, `isSubsumed` performs subsumption test with the input *SemanticDescription*. (iii) **Concept**: models an atomic concept A_i in C_{CN}; `name` contains the concept name, while `denied`, if set to *true*, allows to express $\neg A_i$. (iv) **GreaterThanRole** (respectively **LessThanRole**): models number restrictions in C_{GT} and C_{LT}. Role name and cardinality are stored in the homonym variables. (v) **UniversalRole**: a universal restriction $\forall R.D$ belonging to $C_{\forall}$; R is stored in `name`, while D is a *SemanticDescription* instance.

In the last classes, the `equals` method, inherited from *java.lang.Object*, has been overridden in order to properly implement logic-based comparison.

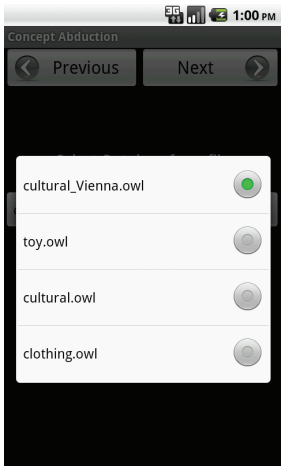
4 Case Study

In order to evidence how the proposed micro-matchmaker works, in this section a toy example is presented in the cultural heritage tourism field. The prototypical reasoner is queried via an ad-hoc GUI client. In what follows, for each provided reasoning task, the interaction the user carries out is shown and obtained results are also displayed as outcomes.

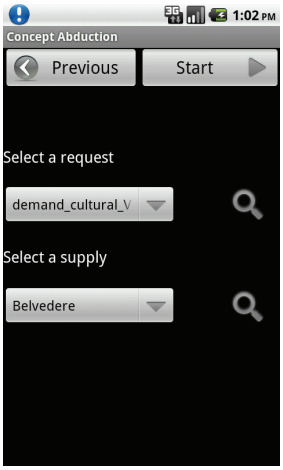
Reasoning task management. First of all the user must select one of the available reasoning tasks and tap *Next* button (see Figure 3(a)). Afterward, the ontology selection screen is shown (Figure 3(b)) listing managed ontologies. In the case under consideration, the user selects the *Cultural Heritage* one.

Concept Abduction. Among possible inferences, the Concept Abduction requires to select both the request R and the supplied resource S to be matched with R (see Figure 3(c)). Request details can be built by browsing the ontology as in Figure 3(d): a graphical and hierarchical view has been preferred hiding

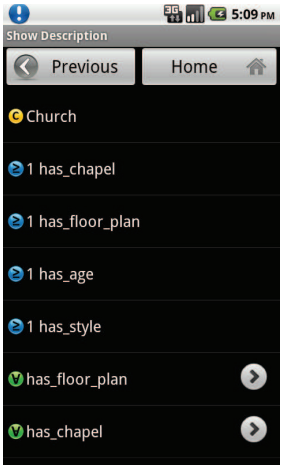
(a) Task selection



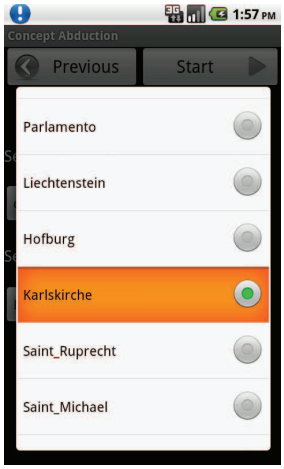
(b) Ontology selection



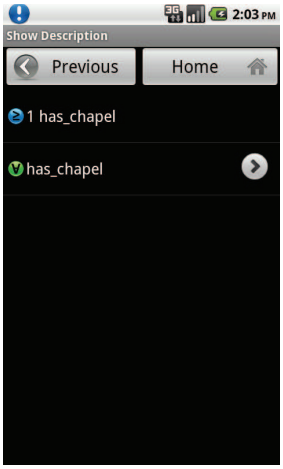
(c) Request/resource selection



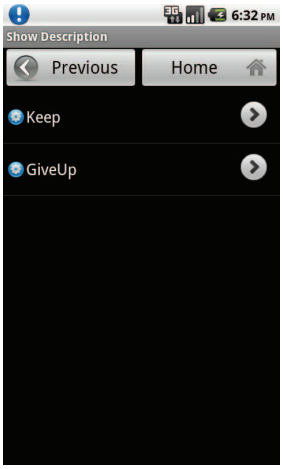
(d) Browse annotation



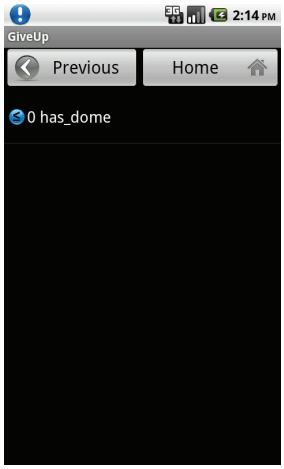
(e) Resource list



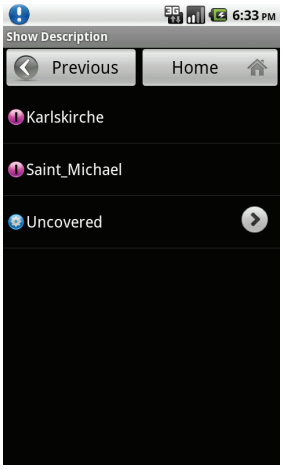
(f) Abduction results



(g) Contraction results



(h) Contraction Give Up



(i) Covering results

Fig. 3. Client application screenshots

the underlying logic-based formalisms. Supplied resources can be picked from the ABox. They are partially shown in Figure 3(e), and reported hereafter:

$(\mathbf{S_1})$**Karlskirche** $\sqsubseteq Church \sqcap \exists\, hasStyle \sqcap \forall\, hasStyle.(Baroque) \sqcap \exists\, hasAge \sqcap \forall\, hasAge.(ModernAge) \sqcap \exists\, hasFloorPlan \sqcap \forall\, hasFloorPlan.(Elliptical) \sqcap (=\ 2\ hasPillar) \sqcap \forall\, hasPillar.(\forall\, hasPosition.(Lateral)) \sqcap \exists\, hasDome \sqcap \forall\, hasDome.(\forall\, hasStyle.(Romanic) \sqcap \forall\, hasFloorPlan.(Elliptical)) \sqcap \exists\, hasMaterial \sqcap \forall\, hasMaterial.(Marble)$.

$(\mathbf{S_2})$**SaintMichael** $\sqsubseteq Church \sqcap \exists\, hasStyle \sqcap \forall\, hasStyle.(Romanesque \sqcap Gothic) \sqcap \exists\, hasAge \sqcap \forall\, hasAge.(MiddleAge) \sqcap (=\ 2\ hasAisle) \sqcap \exists\, hasChapel \sqcap \forall\, hasChapel.(\forall\, hasPosition.(Lateral)) \sqcap \exists\, hasAltar \sqcap \forall\, hasAltar.(\forall\, hasStyle.(Baroque)) \sqcap (=\ 1\ hasApse) \sqcap (=\ 2\ hasCrypt)$.

$(\mathbf{S_3})$**Augustinerkirche** $\sqsubseteq Church \sqcap \exists\, hasStyle \sqcap \forall\, hasStyle.(Gothic) \sqcap \exists\, hasAge \sqcap \forall\, hasAge.(MiddleAge) \sqcap (=\ 1\ hasAisle) \sqcap \exists\, hasCrypt \sqcap \forall\, hasCrypt.(Crypt)$.

$(\mathbf{S_4})$**SaintRuprecht** $\sqsubseteq Church \sqcap \exists\, hasStyle \sqcap \forall\, hasStyle.(Baroque \sqcap Romanesque) \sqcap \exists\, hasAge \sqcap \forall\, hasAge.(MiddleAge) \sqcap (=\ 2\ hasAisle) \sqcap \exists\, hasCeiling \sqcap \forall\, hasCeiling.(\forall\, hasMaterial.(Wood)) \sqcap \exists\, hasAltar \sqcap \forall\, hasAltar.(\forall\, hasStyle.(Romanesque)) \sqcap \exists\, hasTower \sqcap \forall\, hasTower.(\forall\, hasStyle.(Romanesque))$.

When the user taps *Start* button the reasoning begins and finally outcomes are portrayed. As an example let us suppose the user is searching for a *modern-age church, built in Baroque style, with an elliptical floor plan and a lateral chapel*. The corresponding formal request, connoted as R_1, is:

$(\mathbf{R_1})$**Request** $\equiv Church \sqcap \exists\, hasStyle \sqcap \forall\, hasStyle.(Baroque) \sqcap \exists\, hasAge \sqcap \forall\, hasAge.(ModernAge) \sqcap \exists\, hasFloorPlan \sqcap \forall\, hasFloorPlan.(Elliptical) \sqcap \exists\, hasChapel \sqcap \forall\, hasChapel.(\forall\, hasPosition.(Lateral))$.

By solving the *Concept Abduction Problem* [23] it is possible to compute an hypothesis H –*i.e.*, the part of the request not fully satisfied by the monument– needed to reach a *full match* with R_1. As depicted in Figure 3(f), the result panel lists all missing features in the *Karlskirche* church (which does not have a lateral chapel) w.r.t. the request:

$\mathbf{H_{(R_1,S_1)}} \equiv \exists\, hasChapel \sqcap \forall\, hasChapel.(\forall\, hasPosition.(Lateral))$.

Concept Contraction. Let us suppose to modify the user request in order to prove the Concept Contraction functionality. The user is now interested in a *modern-age church, built in Baroque style, with an elliptical floor plan and lacking domes*. This request (R_2) can be formally expressed as:

$(\mathbf{R_2})$**Request** $\equiv Church \sqcap \exists\, hasStyle \sqcap \forall\, hasStyle.(Baroque) \sqcap \exists\, hasAge \sqcap \forall\, hasAge.(ModernAge) \sqcap \exists\, hasFloorPlan \sqcap \forall\, hasFloorPlan.(Elliptical) \sqcap (\leq\ 0\ hasDome)$.

In this case, *Karlskirche* is clearly incompatible w.r.t. R_2, so the user can go back to main screen and select the *Concept Contraction* task [23] in order to analyze motivation for that. Result screen is reported in Figure 3(g). When the user selects the *Keep* function, compatible request properties are shown, whereas *Give Up* lists incompatible elements, as in Figure 3(h). Particularly, *Karlskirche* is partially incompatible because it has a dome, despite it satisfies the remaining features. Concept Contraction results formally are:

$\mathbf{G_{(R_2,S_1)}} \equiv (\leq\ 0\ hasDome)$.

$\mathbf{K_{(R_2,S_1)}} \equiv Church \sqcap \exists\, hasStyle \sqcap \forall\, hasStyle.(Baroque) \sqcap \exists\, hasAge \sqcap \forall\, hasAge.(ModernAge) \sqcap \exists\, hasFloorPlan \sqcap \forall\, hasFloorPlan.(Elliptical)$.

Concept Covering. Let us now vary the request one more time to test the Concept Covering : R_3 refers to a *modern-age church, built in Baroque style, with an elliptical floor plan, which includes a dome, built in Byzantine style and with an elliptical floor plan, and a side chapel*. Formally:

$(\mathbf{R_3})$**Request** $\equiv Church \sqcap \exists\, hasStyle \sqcap \forall\, hasStyle.(Baroque) \sqcap \exists\, hasAge \sqcap \forall\, hasAge.(ModernAge) \sqcap \exists\, hasFloorPlan \sqcap \forall\, hasFloorPlan.(Elliptical) \sqcap$

$\forall\ hasChapel.(\ \forall\ hasPosition.(Lateral)) \sqcap \exists\ hasDome \sqcap \forall\ hasDome.(\ \forall\ hasStyle.(Byzantine) \sqcap \forall\ hasFloorPlan.(Elliptical))$.

Running the *Concept Covering* task the mobile matchmaker solves the *Concept Covering Problem (CCoP)* [12]. Results are in Figure 3(i) and include the set of compatible candidates whose intersection covers R_3 as much as possible (*Karlskirche* and *SaintMichael* in our case) along with the uncovered part of the request U. Particularly, neither available monument has a dome built in Byzantine style, so U is (in DL):

$\mathbf{U_{(R_3,S)}} \equiv\ \forall\ hasDome.(\ \forall\ hasStyle.(Byzantine))$.

5 Experiments

A performance evaluation was carried out for non-standard inferences on a PC testbed[3] and on an Android smartphone[4]. The test performs both unfolding and normalization over a 557 kB knowledge base with 100 request/resource pairs, randomly generated starting from the ontology defined in [23], with average size of 4.2 kB and finally executes abduction and contraction between each pair. Every task was repeated four times and the average turnaround time of the last three runs was taken. Figure 4 and Figure 5 report on time results (in microseconds) in case of PC and handheld testbed, respectively.

For each request/resource the system checks for compatibility; in case, the abduction is performed, otherwise it is run the contraction followed by an abduction with the compatible part of the request. Notice that the computational time basically varies depending on the complexity of the semantic descriptions. Results for mobile tests have been referred to the ones for PC in order to highlight non-standard inferences exhibit similar trends. Figure 6 shows processing times in a logarithmic scale: tough the performance gap between fixed and handheld

[3] CPU Intel Core i7 CPU 860 at 2.80 GHz (4 cores/8 threads), 8 GB DDR3-SDRAM (1333 MHz), 1 TB SATA (7200 RPM) hard disk, 64-bit Microsoft Windows 7 Professional and 64-bit Java 7 SE Runtime Environment (build 1.7.0_03-b05).

[4] Samsung Galaxy Nexus GT-I9250 with Dual-core ARM Cortex A9 CPU at 1,2 GHz, 1 GB RAM, 16 GB storage memory, and Android version 4.2.2.

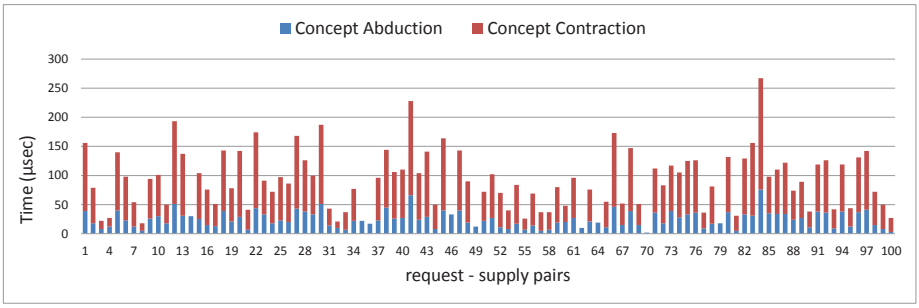


Fig. 4. Turnaround Time on PC

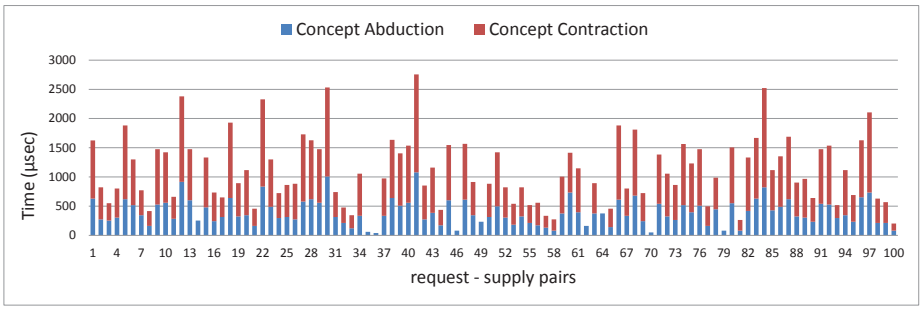


Fig. 5. Turnaround Time on Mobile

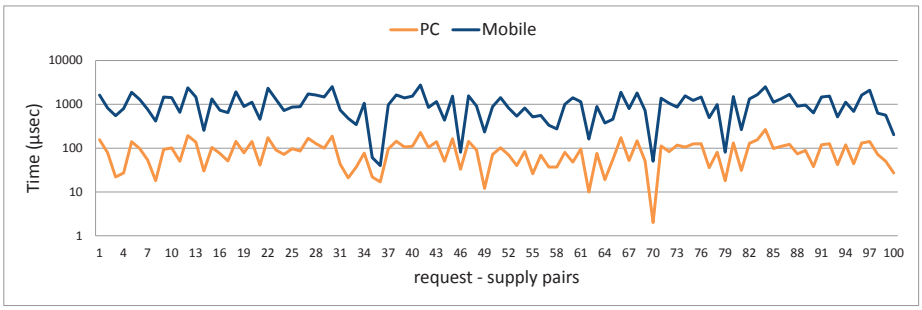


Fig. 6. Total processing time on PC and mobile devices

architectures, reasoning tasks maintain an acceptable computational load also on mobiles. Times are roughly one order of magnitude higher in the Android device. This is due not only to the limited computational capabilities of mobile devices, but also to the harder memory constraints on smartphones which impose to have as much free memory as possible at any time. Consequently, the Android Dalvik virtual machine performs more frequent and heavy garbage collection w.r.t. Java SE virtual machine so increasing the total processing time.

6 Related Work

Logic-based matchmaking requires careful optimization to achieve acceptable performance for adequately expressive languages [1, 2]. This is specifically true in case of inferences performed on mobile computing platforms where an efficient implementation of reasoning services is still an open problem. Several techniques [2] allow to increase expressiveness or decrease running time at the expense of main memory usage. *Pocket KRHyper* [7] was the first reasoning engine specifically designed for mobile devices. It supported the $\mathcal{ALCHIR}+$ DL and was built as a Java ME (Micro Edition) library. However, it presented limitation in size

and complexity of managed logic expressions. To overcome those constraints, tableaux optimizations were introduced in [16] and implemented in *mTableau*, a modified version of Java SE *Pellet* reasoner [13]. Comparative performance tests were performed on a PC, showing faster turnaround times than both unmodified Pellet and *Racer* [3] reasoner. Nevertheless, the Java SE technology is not expressly tailored to the current generation of mobile devices. In fact, other relevant reasoners, such as *FaCT++* [10] and *HermiT* [15], cannot run on common mobile platforms. Porting would require a significant re-write or re-design effort, since they rely on Java class libraries incompatible with most widespread mobile OS (*e.g.*, Android). In latest years, a different approach to implement reasoning tools aimed to simplify both the underlying logic languages and admitted KB axioms. Structural algorithms could be adopted but maintaining expressiveness enough for broad application areas. In [8], the basic $\mathcal{EL}$ DL was extended to $\mathcal{EL}^{++}$, a language deemed suitable for various applications, characterized by very large ontologies with moderate expressiveness. A structural classification algorithm was also devised, which allowed high-performance $\mathcal{EL}^{++}$ ontology classifiers such as CEL [9], ELK [28] and Snorocket [20]. OWL 2 profiles definition complies with this perspective, focusing on language subsets of practical interest for important application areas rather than on fragments with significant theoretical properties. Other mobile engines currently provide rule processing for entailments materialization in a KB [24, 21, 27], but available features are not suitable to support applications requiring non-standard inference tasks and extensive reasoning over ontologies [27]. Standard inference services, such as satisfiability and subsumption, only provide binary "yes/no" answers. Consequently, they can only distinguish among *full* (*subsume*), *potential* (*intersection-satisfiable*) and *partial* (*disjoint*) match types (adopting the terminology in [11] and [5], respectively). Non-standard inferences, as Concept Abduction, Contraction and Covering, are needed to enable a more fine-grained semantic ranking as well as explanations of outcomes [11]. In [17] an early approach was proposed to adapt non-standard logic-based inferences to pervasive computing contexts. By limiting expressiveness to $\mathcal{AL}$ language, acyclic, structural algorithms were adopted reducing standard and non-standard inferences to set-based operations [14]. KB management and reasoning were then executed through a data storage layer, based on a mobile Relational DBMS (RDBMS). Such an approach was further investigated in [22] and [23], by increasing the expressiveness to $\mathcal{ALN}$ DL and allowing larger ontologies and more complex descriptions, through the adoption of mobile Object-Oriented DBMS (OODBMS).

7 Conclusion and Future Work

The paper presented a prototypical reasoner for pervasive computing. It supports Semantic Web technologies through the OWL API and implements both standard and non-standard inferences. Developed in Java, it targets the Android platform. Early experiments were made both on PCs and smartphones and evidenced competitiveness with FaCT++, HermiT and Pellet reasoners in standard

inferences. Besides further performance optimization leveraging Android Dalvik peculiarities, future work includes: support for ABox management and OWLlink protocol[5], $\mathcal{EL}^{++}$ extension of abduction and contraction.

Acknowledgments

The authors acknowledge partial support of EU PO Apulia region FESR projects "UbiCare" and "Lean Software Development".

References

1. Baader, F. and Hollunder, B. and Nebel, B. and Profitlich, H.J. and Franconi, E.: An empirical analysis of optimization techniques for terminological representation systems. 4(2), 109–132 (1994)
2. Horrocks, I. and Patel-Schneider, PF: Optimizing description logic subsumption. 9(3), 267–293 (1999)
3. Haarslev, V. and Müller, R.: Racer system description. pp.701–705 (2001)
4. F. Baader and D. Calvanese and D. Mc Guinness and D. Nardi and P. Patel-Schneider: The Description Logic Handbook. Cambridge University Press (2002)
5. Li, L. and Horrocks, I.: A software framework for matchmaking based on semantic web technology. 8(4), 39–60 (2004)
6. ITU: Internet Reports 2005: The Internet of Things (November 2005)
7. A. Sinner and T. Kleemann: KRHyper - In Your Pocket. pp. 452–457. Tallinn, Estonia (July 2005)
8. Baader, F. and Brandt, S. and Lutz, C.: Pushing the EL envelope. vol. p., 364. Lawrence Erlbaum Associates LTD (2005)
9. Baader, F. and Lutz, C. and Suntisrivaraporn, B.: CEL – a polynomial-time reasoner for life science ontologies. pp.287–291 (2006)
10. Tsarkov, D. and Horrocks, I.: Fact++ description logic reasoner: System description. pp.292–297 (2006)
11. Colucci, Simona and Di Noia, Tommaso and Pinto, Agnese and Ragone, Azzurra and Ruta, Michele and Tinelli, Eufemia: A Non-Monotonic Approach to Semantic Matchmaking and Request Refinement in E-Marketplaces. 12(2), 127–154 (2007)
12. Azzurra Ragone and Tommaso Di Noia and Eugenio Di Sciascio and Francesco M. Donini and Simona Colucci and Francesco Colasuonno: Fully Automated Web Services Discovery and Composition through Concept Covering and Concept Abduction. 4(3), 85–112 (2007)
13. Sirin, E. and Parsia, B. and Grau, B.C. and Kalyanpur, A. and Katz, Y.: Pellet: A practical OWL-DL reasoner. 5(2), 51–53 (2007)
14. Di Noia, Tommaso and Di Sciascio, Eugenio and Donini, Francesco M.: Semantic matchmaking as non-monotonic reasoning: A description logic approach. 29, 269–307 (2007)
15. Shearer, R. and Motik, B. and Horrocks, I.: Hermit: A highly-efficient owl reasoner. pp. 26–27 (2008)

[5] OWLlink Structural Specification, W3C Member Submission, 1 July 2010, htp://www.w3.org/Submission/owllink-structural-specification/

16. Steller, L. and Krishnaswamy, S.: Pervasive Service Discovery: mTableaux Mobile Reasoning. In: Int. Conf. on Semantic Systems (I-Semantics). Graz, Austria (2008)
17. Ruta, Michele and Di Noia, Tommaso and Di Sciascio, Eugenio and Piscitelli, Giacomo and Scioscia, Floriano: A semantic-based mobile registry for dynamic rfid-based logistics support. pp. 1–9. ACM, New York, USA (2008)
18. Horridge, M. and Bechhofer, S.: The OWL API: a Java API for working with OWL 2 ontologies. 2009 (2009)
19. Moguillansky, M. and Wassermann, R. and Falappa, M.: An argumentation machinery to reason over inconsistent ontologies. pp.100–109 (2010)
20. Lawley, M.J. and Bousquet, C.: Fast classification in Protégé: Snorocket as an OWL 2 EL reasoner. vol. pp., 45–49 (2010)
21. Kim, T. and Park, I. and Hyun, S.J. and Lee, D.: MiRE4OWL: Mobile Rule Engine for OWL. pp. 317–322. IEEE (2010)
22. Michele Ruta and Floriano Scioscia and Eugenio Di Sciascio: 18th Italian Symposium on Advanced Database Systems (SEBD2010). pp. 210–221. Esculapio (2010)
23. Ruta, M. and Di Sciascio, E. and Scioscia, F.: Concept abduction and contraction in semantic-based P2P environments. 9(3), 179–207 (2011)
24. Tai, W. and Keeney, J. and O'Sullivan, D.: COROR: a composable rule-entailment owl reasoner for resource-constrained devices. pp.212–226 (2011)
25. M. Ruta and F. Scioscia and E. Di Sciascio and G. Loseto: Semantic-based Enhancement of ISO/IEC 14543-3 EIB/KNX Standard for Building Automation. 7(4), 731–739 (2011)
26. Michele Ruta and Floriano Scioscia and Giuseppe Loseto and Filippo Gramegna and Agnese Pinto and Saverio Ieva and Eugenio Di Sciascio: 5th International Workshop on Semantic Sensor Networks. Proc. of the 11th International Semantic Web Conference. vol. pp., 17–32 (2012)
27. Motik, B. and Horrocks, I. and Kim, S.M.: Delta-Reasoner: a Semantic Web Reasoner for an Intelligent Mobile Platform. pp. 63–72. ACM, New York, NY, USA (2012)
28. Kazakov, Yevgeny and Krötzsch, Markus and Simančík, František: OWL Reasoner Evaluation Workshop (ORE 2012). vol. pp., 106–117. CEUR-WS (2012)
29. Michele Ruta and Floriano Scioscia and Filippo Gramegna and Giuseppe Loseto and Eugenio Di Sciascio: 20th Italian Symposium on Advanced Databases Systems (SEBD 2012). pp. 289–294. Edizioni Libreria Progetto (Jun 2012)
30. Michele Ruta and Floriano Scioscia and Saverio Ieva and Giuseppe Loseto and Eugenio Di Sciascio: Semantic Annotation of OpenStreetMap Points of Interest for Mobile Discovery and Navigation. pp. 33–39. IEEE (2012)