Mini-ME Swift: The First Mobile OWL Reasoner for iOS

Michele Ruta(✉), Floriano Scioscia, Filippo Gramegna, Ivano Bilenchi, and Eugenio Di Sciascio

Polytechnic University of Bari, via E. Orabona 4, 70125 Bari, Italy
{michele.ruta,floriano.scioscia,filippo.gramegna,
ivano.bilenchi,eugenio.disciascio}@poliba.it

Abstract. Mobile reasoners play a pivotal role in the so-called Semantic Web of Things. While several tools exist for the Android platform, iOS has been neglected so far. This is due to architectural differences and unavailability of OWL manipulation libraries, which make porting existing engines harder. This paper presents Mini-ME Swift, the first Description Logics reasoner for iOS. It implements standard (Subsumption, Satisfiability, Classification, Consistency) and non-standard (Abduction, Contraction, Covering, Difference) inferences in an OWL 2 fragment. Peculiarities are discussed and performance results are presented, comparing Mini-ME Swift with other state-of-the-art OWL reasoners.

1 Introduction and Motivation

Semantic Web technologies have been increasingly adopted in resource-constrained volatile environments through the *Semantic Web of Things* (SWoT) [8,25], whose goal is embedding intelligence in pervasive contexts. Semantic Web languages underlie knowledge representation and interoperability in the SWoT, particularly the Resource Description Framework (RDF)[1] and the Web Ontology Language (OWL)[2]. Anyway, although mobile devices are even more powerful, porting existing Semantic Web inference engines to mobile operating systems is not a straightforward task and it may result in suboptimal performance. Among the challenges for the next decade of the Semantic Web, Bernstein *et al.* [3] outlined *"languages and architectures that will provide [...] knowledge to the increasingly mobile and application-based Web"*. More specifically, Yus and Pappachan [37] pointed out the *"lack of semantic reasoners for certain mobile operating systems (such as iOS)"* among the problems of developing semantic mobile apps. Several solutions are currently available for Android [4], due to its support for the Java language, allowing to port popular OWL manipulation

[1] RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation 25 February 2014, https://www.w3.org/TR/rdf11-concepts/.

[2] OWL 2 Web Ontology Language Document Overview (Second Edition), W3C Recommendation 11 December 2012, https://www.w3.org/TR/owl2-overview/.

P. Hitzler et al. (Eds.): ESWC 2019, LNCS 11503, pp. 298–313, 2019.
https://doi.org/10.1007/978-3-030-21348-0_20

libraries. Conversely, the Apple iOS mobile platform has been neglected so far, due to differences in architecture and development tools.

Hence, this paper introduces *Mini-ME Swift*, the first Description Logics reasoner and matchmaker for iOS. It has been developed in Swift 4 language aiming to the Application Programming Interface (API) parity with the Java-based *Mini Matchmaking Engine* [26], though it has been designed and implemented from scratch to achieve significantly better performance. Logic expressiveness is limited to an OWL fragment corresponding to the $\mathcal{ALN}$ (Attributive Language with unqualified Number restrictions) Description Logic (DL) on acyclic Terminological boxes (TBoxes). It efficiently implements standard (Subsumption, Satisfiability, Classification, Consistency) and non-standard (Abduction, Contraction, Covering, Difference) inferences. A case study on semantic-enhanced Point of Interest (POI) discovery in Mobile Augmented Reality (MAR) has allowed validating the effectiveness and ease of integration of the proposed matchmaker in iOS apps. Architectural and optimization solutions have been assessed in an experimental campaign, comparing Mini-ME Swift with the Java-based Mini-ME as well as with four popular Semantic Web reasoners. Inference time and memory usage are reported on a conventional desktop testbed, and required computational resources have been evaluated on current mobile devices.

The remainder of the paper is as follows. Section 2 recalls essential background information and relevant related work. Mini-ME Swift design and optimization strategies are discussed in Sect. 3. Section 4 presents the case study, while performance assessment is in Sect. 5, before conclusions.

2 Background

Related Work. Due to architectural constraints and computational complexity of Description Logics reasoning, the majority of early mobile inference engines provided only rule processing for entailment materialization in a Knowledge Base (KB); proposals include *3APL-M* [11], *COROR* [31], *MiRE4OWL* [10], *Delta-Reasoner* [17] and the system in [27]. The μOR reasoner [1] adopts a resolution algorithm on the *OWL-Lite*$^-$ language, while *LOnt* [12] works on *DL Lite*, a subset of OWL-Lite. The mobile OWL2 RL engine in [36] exploits an optimization of the classic *RETE* algorithm for rule systems. More expressive DLs were supported by exploiting tableau algorithms: *Pocket KRHyper* [28] adopted the $\mathcal{ALCHIR}+$ DL, but memory limitations curbed the size and complexity of manageable KBs. Tableaux optimization was exploited in *mTableaux* [30] to reduce memory consumption. Fuzzy $\mathcal{ALN}$(D) was supported in [21] via structural algorithms. Android is currently the most widespread mobile platform and Java is its primary application development language. The majority of state-of-the-art OWL reasoners runs on Java Standard Edition (SE) [14], mature Semantic Web language manipulation libraries are available such as *Jena* [15] and the *OWL API* [7]. Anyway, porting existing systems requires significant effort, due to architectural differences and incomplete support of Java SE class libraries under Android. Bobed *et al.* [4] met such barriers when porting *Hermit* [6],

JFact (a Java variant of *Fact++* [34]) and three other OWL reasoners to Android. Jena was ported by *AndroJena*[3], albeit with deep re-design and feature limitations. The *ELK* reasoner has an Android port [9] as well. To the best of our knowledge, no previous reasoner has supported iOS, preventing a relevant segment of users and application developers from exploiting semantic technologies effectively. In fact, a few semantic-based iOS apps and prototypes do exist, but they either exploit remote servers for reasoning [19,23,35] or precompute inferences on a conventional computer, storing results on the mobile device [20]. The iOS port of a subset of the OWL API [22] has enabled the development of Mini-ME Swift. From a performance optimization standpoint, *TrOWL* [33], *Konclude* [29] and other recent inference engines implement multiple different techniques and then select the best one according to the logical expressiveness of the particular KB and/or the required inference task. Konclude can also exploit parallel execution on multi-core processors and it has been the top performer in latest OWL DL reasoner competitions [18]. Nevertheless, SWoT application surveys [5,13] evidence inherently unpredictable contexts requiring mobile agents endowed with quick decision support, query answering and stream reasoning capabilities. Specific non-standard inference services may be more suitable than standard ones in those cases [26].

Inference Services. The proposed reasoner leverages polynomial-complexity structural algorithms exploiting KB preprocessing stages for concept *unfolding* and *Conjunctive Normal Form* (CNF) normalization through recursive procedures. In $\mathcal{ALN}$ CNF, every concept expression is either $\bot$ (*Bottom* a.k.a. *Nothing*) or the conjunction ($\sqcap$) of: (possibly negated) atomic concepts; greater-than ($\geq$) and less-than ($\leq$) number restrictions, no more than one per type per role; universal restrictions ($\forall$), no more than one per role, with filler recursively in CNF. As said, Mini-ME Swift supplies *standard Subsumption* and *Satisfiability*. In advanced scenarios they are not enough, as they provide only a Boolean answer. Therefore, *non-standard Concept Abduction* (CA) and *Concept Contraction* (CC) allow extending, respectively, (missed) subsumption and (un)satisfiability, in the Open World Assumption [26]. Based on CNF norm, they also provide a *penalty* metric evidencing a semantic distance ranking of KB instances (a.k.a. resources in matchmaking settings) w.r.t. a target individual (a.k.a. request). Mini-ME Swift further includes *Concept Difference* (CD) [32] to subtract information in a concept description from another one. While CA, CC and CD are useful in one-to-one discovery, matchmaking and negotiation scenarios, Mini-ME also includes the *Concept Covering Problem* (CCoP) non-standard inference for many-to-one composition of a set of elementary instances to answer complex requests [26]. Mini-ME Swift can be also exploited in more general knowledge-based applications, as it provides *Coherence* and *Classification* services over ontologies: Coherence is close to *Ontology Satisfiability*, but it does not process individuals [16]; Ontology Classification computes the overall concept taxonomy induced by the subsumption relation, from $\top$ (*Top* a.k.a. *Thing*) to $\bot$.

[3] AndroJena project page: https://github.com/lencinhaus/androjena.

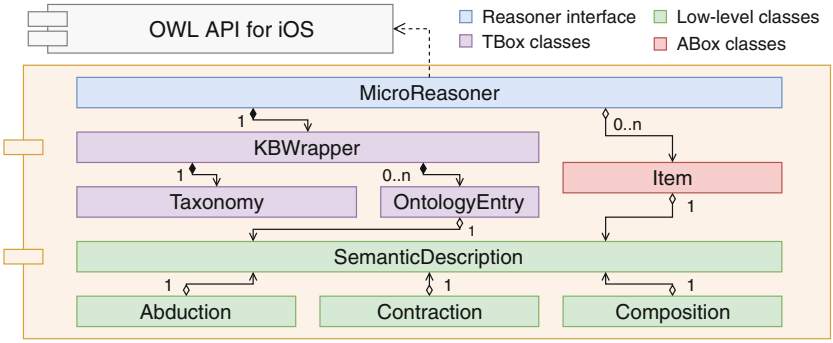


Fig. 1. Reasoner architecture

3 Description Logics Reasoning for iOS Devices

Reasoner Architecture. The proposed tool is written in Swift 4. It can be compiled as *iOS Framework*, *i.e.*, as dynamic linking library. It targets all iOS devices running system version 8 or later, and it also runs unmodified on macOS 10.11 and later. By leveraging the expressive power of the Swift language – particularly its functional features, such as *optionals*, *optional binding* and *higher order functions*[4] – the codebase is still compact, with a good readability. The reasoner architecture, in Fig. 1, has been kept purposefully simple to avoid unnecessary layers of abstraction and the associated overhead. Main components are described hereafter. Particularly, the **KBWrapper** and **SemanticDescription** classes implement most of the available reasoning tasks on ontologies and concept descriptions, respectively. The **MicroReasoner** class acts as a facade hiding the interactions between lower-level components. In detail:

- **OWL API for iOS:** is the port [22] of the OWL API to the iOS platform, providing support for parsing and manipulating OWL2 KBs in $\mathcal{ALEN}$ (Attributive Language with unqualified Existential and Number restrictions) DL with RDF/XML syntax.
- **MicroReasoner:** is the library entry point, exposing KB operations such as parsing the target ontology and loading/fetching instances to be used for matchmaking. It also exposes standard and non-standard inferences.
- **KBWrapper:** allows KB management, *i.e.*, creation of internal data structures and concept unfolding as well as Classification and Coherence check on ontologies.
  Storage, preprocessing and inference procedures on concept expressions are implemented by methods of different high-level data structures owned by the *KBWrapper*:

[4] Swift documentation: https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language.

- **OntologyEntry:** when parsing an ontology, the KBWrapper loads the TBox, whose concepts and associated descriptions are stored as (*IRI*, *OntologyEntry*) pairs. *OntologyEntry* instances encapsulate the information about any concept involved in a reasoning task, *e.g.*, its expression and its normalization cache value (see Sect. 3).
- **Taxonomy:** models the concept hierarchy as resulting from Classification. It allows manipulating the tree, merging equivalent nodes, and retrieving ancestors or successors of a given node.
- **Item:** represents matchmaking resources and requests, but it can refer to any named concept expression. It is composed by an *IRI* and a *SemanticDescription*.

TBox and Assertion Box (ABox) manipulation and reasoning are supported by the following low-level data structure layer:

- **SemanticDescription:** models an $\mathcal{ALN}$ concept expression in CNF as conjunction of C_{CN}, $C_{\geq}$, $C_{\leq}$, $C_{\forall}$ components, stored in collections of *AtomicConcept*, *GreaterThanRole*, *LessThanRole* and *UniversalRole* Swift class instances, respectively.
- **Abduction, Contraction, Composition:** model the results of CA, CC and CCoP, respectively. *Abduction* and *Contraction* include a penalty score [26].

Optimization Strategies. Some of the lower-level optimizations in Mini-ME Swift have been achieved thanks to specific characteristics and implementation details of the programming language. The implicit use of *Copy on Write* (CoW) for Swift collections –transparent to the developer– has enabled a straightforward performance improvement in algorithms heavily relying on conditionally mutating collections (concept unfolding and normalization). Further benefits stem from the adoption of *structs* instead of classes for some of the lower-level data structures, since they are often allocated on the stack rather than on the heap. However, *Automatic Reference Counting* (ARC)[5] –the memory management strategy employed by the Swift compiler– sometimes would have led to performance degradation in critical code sections. As an example, one of the recursive tree traversal algorithms implemented in Mini-ME Swift originally spent about 80% of its total CPU time in *retain* and *release* function calls on the nodes of the tree. This has been addressed by leveraging *unmanaged references*[6] wherever needed. High-level optimization has also been carried out in order to improve both inference turnaround time and memory usage. Enhancements concern:

- **Ontology loading and preprocessing:** once Mini-ME Swift is instantiated, the target ontology is loaded into the internal data structures and preprocessed. At this stage, terminological equivalences are transitively unfolded and merged in single entries (*e.g.*, if $C \equiv D$ and $D \equiv E$, then $C \equiv D \equiv$

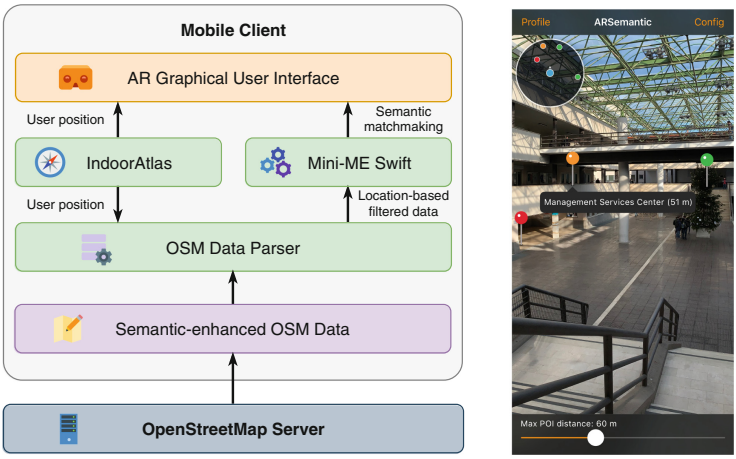
[5] Automatic reference counting documentation: https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html.

[6] Unmanaged references documentation: https://developer.apple.com/documentation/swift/unmanaged.

$E \equiv C \sqcap D \sqcap E$), saving memory and processing time. Furthermore, *told subsumption cycles* [34] are identified and marked in order to be efficiently solved while classifying. These are the only possible cyclic references supported by Mini-ME Swift. Finally, concept and role names are translated to internal numerical identifiers, allowing for more efficient storage and processing. In particular, memory addresses are exploited as IDs, leveraging the uniqueness of IRI in-memory instances obtained from the OWL API iOS port [22].

- **Concept unfolding and normalization:** the *unfolding* and *CNF normalization* algorithms were initially optimized by caching completely unfolded and normalized concepts. However, it soon became evident that caching could be extended to *intermediate unfolded concepts* as well: given an acyclic concept B, every other concept C recursively unfolded as part of the unfolding of B is also completely unfolded, making it suitable for caching. However, C is not yet in normal form, therefore the concept cache must keep track of whether stored concepts have been only unfolded or both unfolded and normalized. This strategy has two benefits: (i) it enables the reuse of the unfolded description of C as part of the normalization of further concepts; (ii) it minimizes computation when C needs to be normalized, since the unfolding step is executed just once.
- **Classification:** Mini-ME Swift adopts a variant of the *enhanced traversal* algorithm in [2], which also accounts for subsumption cycles detected while preprocessing the ontology. Subsumption check results are *cached*, as customary for OWL reasoners, though significant effort went in ensuring checks are avoided whenever possible: classification is performed according to the concept *definition order* [2], which allows skipping the *bottom search* step for primitive concepts having acyclic descriptions. The exploitation of *told disjoints* [34] and *told subsumers* has been implemented. Moreover, *synonym merging* (*e.g.*, if it is inferred that $B \equiv C$, then the taxonomy nodes for B and C are merged) reduces memory usage and search time by making the tree smaller.
- **Ontology Coherence:** a naive approach involves performing CNF normalization for every concept in the TBox [26]. However, earlier experimental results showed this process is time consuming (particularly for larger TBoxes). This could be smoothed by adopting forceful caching policies for unfolded concepts, albeit paying additional occupancy of memory. Mini-ME Swift sidesteps this problem by computing the Coherence check through a modified version of the Classification algorithm, which stops as soon as an unsatisfiable concept is detected. Since normalization is *lazy* –*i.e.*, it is computed only when needed for an inference task– and Classification explicitly avoids subsumptions as much as possible, the overall number of performed normalizations is also reduced. This results in significantly improved time and memory efficiency w.r.t. the naive approach. Furthermore, the Coherence check of a TBox $\mathcal{T}$ can be skipped in the following cases: (i) $\mathcal{T}$ is *trivially incoherent* if $\exists$ a concept expression C in $\mathcal{T} \mid C \sqsubseteq \bot$; (ii) $\mathcal{T}$ is *trivially coherent* if $\mathcal{T}$ contains no *disjoint concept* axioms and *number restrictions* are either absent or all of the same type (*i.e.*, either minimum or maximum cardinality).

Both conditions can be verified inexpensively while loading the KB, since this only entails checking whether given constructors are in the TBox.



(a) Component diagram with data flow (b) User interface

Fig. 2. MAR POI semantic discovery prototype

4 Case-Study: Semantic-Based POI Discovery in Augmented Reality

In order to validate the effectiveness and the ease of use of Mini-ME Swift in ubiquitous computing applications, the existing Android framework in [24] has been rewritten in Swift as a prototypical iOS app. The tool leverages crowd-sourced OpenStreetMap[7] (OSM) cartography, annotated exploiting Semantic Web technologies to provide POIs with rich structured descriptions. Figure 2a illustrates application components. The mobile client communicates with an OSM server, providing map elements enriched with formal machine-understandable metadata. The app uses Mini-ME Swift to execute semantic matchmaking [26] between an annotated user profile and POIs in the area surrounding user's location. Position is obtained from embedded smartphone devices, including accelerometer, compass and Wi-Fi trilateration. *IndoorAtlas*[8] toolkit, relying on the device magnetometer, provides accurate localization in indoor contexts. As shown in Fig. 2b, semantic matchmaking outcomes are displayed as color-coded markers in a MAR graphical UI leveraging *ARKit*[9] iOS library. Colors from green to red

[7] OpenStreetMap: https://www.openstreetmap.org/.
[8] IndoorAtlas: https://www.indooratlas.com/.
[9] Apple ARKit 2: https://developer.apple.com/arkit/.

represent semantic distance, from full to partial matches. The user can touch a marker to access result explanation, in terms of missing and/or conflicting characteristics between her profile and the POI. Due to lack of space, the reader is referred to [24] for details on the map annotation method and POI discovery algorithm. By following iOS developer guidelines, the integration of Mini-ME Swift, IndoorAtlas and ARKit has been straightforward.

Table 1. Dataset-wide times and min/max memory peak for Classification

		Fastest in # tests	Parsing Time (s)	Classification time (s)	Min Mem. Peak (MB)	Max Mem. Peak (MB)
Reasoners running on Mac Pro	Fact++	0	1507.92	430.61	81.00	3330.00
	HermiT	0	614.57	682.95	62.49	8547.68
	Konclude	351	132.04	89.60	14.10	3797.46
	TrOWL	0	621.35	401.79	54.74	4427.46
	Mini-ME Java	2	629.96	9841.24	58.15	11854.69
	Mini-ME Swift	948	209.10	60.28	5.92	1546.07
Mini-ME Swift on mobile devices	iPhone 7		180.15	83.78	15.11	1198.80
	iPad Mini 4		411.42	191.38	12.88	1182.57
	iPhone 5s		539.17	292.84	9.56	449.08

5 Experiments

An experimental evaluation of the proposed system has been carried out on desktop and mobile devices, for standard and non-standard inference services, by means of a bespoke test framework[10]. Desktop tests have been performed on a 2009 Mac Pro[11], while mobile experiments on an iPhone 7[12], an iPad Mini 4[13] and an iPhone 5s[14]. Correctness and completeness of inference services have been checked by comparing obtained results with other state-of-the-art reasoners, used as test oracles. Outcomes returned by Mini-ME Swift have been considered correct in case of unanimous match with all the oracles and incorrect

[10] Code and instructions available under the Eclipse Public License 1.0 at https://github.com/sisinflab-swot/owl-reasoner-test-framework.

[11] Dual Intel Xeon 5500 quad-core CPUs at 2.26 GHz clock frequency, 32 GB DDR3 RAM at 1066 MHz, 640 GB 7200 RPM HDD, OS X 10.11.6 El Capitan.

[12] Apple A10 CPU (2 high-performance cores at 2.34 GHz and 2 low-energy cores), 2 GB LPDDR4 RAM, 32 GB flash storage, iOS 10.1.1.

[13] Dual-core 1.5 GHz Apple A8 CPU, 2 GB LPDDR3 RAM, 16 GB flash storage, iOS 9.3.3.

[14] Dual-core 1.3 GHz Apple A7 CPU, 1 GB LPDDR3 RAM, 16 GB flash storage, iOS 10.3.2.

in case of no match; manual investigation has been performed for partial matches. Performance evaluation has been focused on turnaround time and peak memory usage for each inference task: results are the average of five repeated runs, with a timeout of 30 min for each run. Reported memory peak values represent the maximum resident set size (*MRSS*) for the process, extracted via *BSD time* on Mac, and equivalently through the *getrusage* POSIX call on mobile. The reader can refer to the Mini-ME Swift Web page[15] for getting the reasoner and in-depth documentation on how to reproduce the benchmarks reported hereafter.

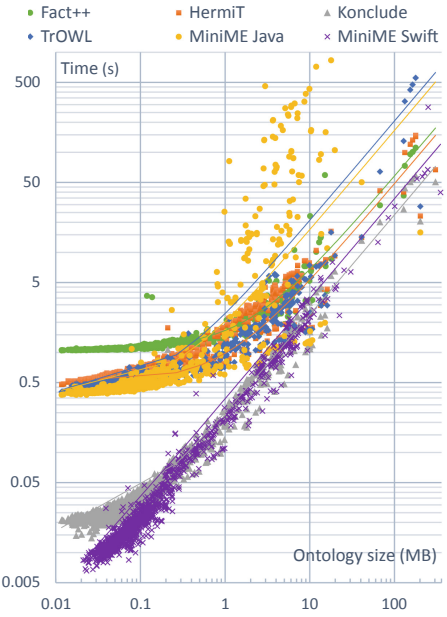
5.1 Standard Inference Services

Ontology Classification and **Coherence** have been evaluated on a set of 1364 KBs, obtained from the 2014 *OWL Reasoner Evaluation Workshop* competition reference dataset[16] considering all the KBs supported by Mini-ME Swift (*e.g.*, having at most $\mathcal{ALN}$ as indicated expressiveness, without general concept inclusions and other unsupported logic constructors). The following reasoners have been used as test oracles of correctness and completeness, as well as references for performance comparison: Fact++ (version 1.6.5) [34], HermiT (1.3.8) [6], Konclude (0.6.2-544) [29], TrOWL (1.5) [33] and Mini-ME Java (2.0) [26]. Before presenting the results of the experimental campaign, it is important to point out that:

- Since Konclude can only parse ontologies in OWL functional syntax and Mini-ME Swift currently only supports the RDF/XML serialization, RDF/XML parsing has been restricted to Mini-ME Swift only, and every other reasoner has been configured to use the functional syntax. This is a worst-case scenario w.r.t. turnaround time, since in general the functional representation is shorter and easier to parse than the equivalent RDF/XML. This is evident in Fig. 3 particularly for the smallest and the largest ontologies, where data points for Mini-ME Swift appear shifted to the right.
- Mini-ME does not process the ABox when computing standard inferences, therefore Ontology Coherence has been informally compared to Consistency as supported by other reasoners, and mismatches have been manually analyzed in order to evaluate the correctness of the reasoner output.
- Since the coherence check is based on a modified version of the Classification algorithm (as stated in Sect. 3), the Ontology Coherence performance results are similar to Classification, hence they have not been reported here.

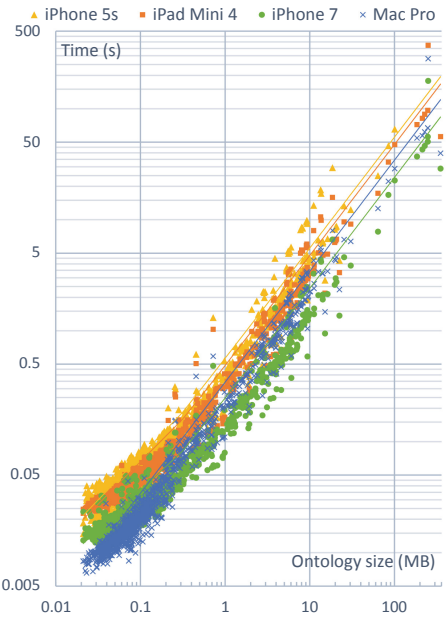
Correctness. Mini-ME Swift has classified all the 1364 ontologies correctly. The Coherence test has returned matching results for 1353 ontologies (99.2% of the dataset). The remaining 11 ontologies contain unsatisfiable classes with no instances, resulting in being considered incoherent by Mini-ME but consistent by other reasoners.

[15] http://sisinflab.poliba.it/swottools/minime-swift.
[16] http://dl.kr.org/ore2014.

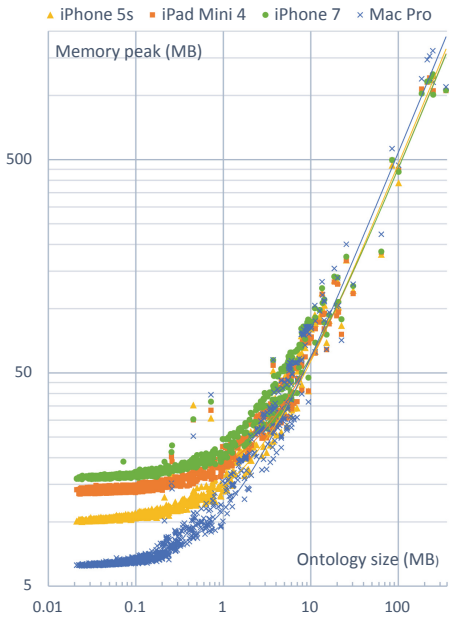


(a) Desktop processing time by ontology size

(b) Desktop memory peak by ontology size



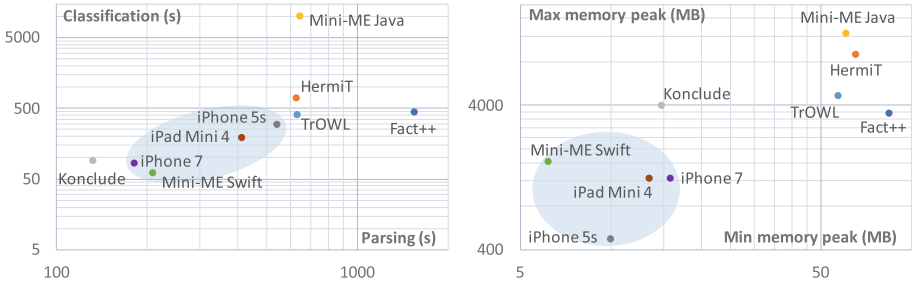
(c) Mobile processing time by ontology size



(d) Mobile memory peak by ontology size

Fig. 3. Performance results for the classification task on desktop and mobile devices

Desktop Performance. Figure 3a plots Classification turnaround times as a function of the ontology size. The time axis scale is limited to 600 s since all reasoners are able to process each ontology in the dataset well within the imposed timeout; the only exceptions are Mini-ME Java and TrOWL, which have reached the timeout on 8 and 1 ontologies, respectively. Mini-ME Swift outperformed all reference systems for ontologies smaller than 200 kB ca.; for larger input, Konclude and Mini-ME Swift have similar performance, both substantially faster than the other reasoners. Although Mini-ME Swift has the advantage of focusing on a smaller OWL2 fragment than the other reasoners, results are influenced by parsing, where Mini-ME Swift is disadvantaged due to RDF/XML adoption. Table 1 displays the cumulative classification performance on the 1301 ontologies correctly classified by all systems within the timeout, and the number of cases where each reasoner was the fastest one; Mini-ME Swift exhibits the lowest overall time by a significant margin, if parsing is factored out. Figure 3b shows the peak memory usage: it is consistently and significantly lower for Mini-ME Swift than the reference tools. Specifically, as reported in Table 1, both the minimum and maximum peaks over the dataset are at least 50% lower than the next-best result for Classification.



(a) Parsing and Classification: total time (b) Classification: min/max memory peak

Fig. 4. Performance statistics for classification on both desktop and mobiles

Mobile Performance. Mini-ME Swift has successfully executed the standard inferences on the same dataset of desktop testbed for all tested mobiles except iPhone 5s, where it failed to process KBs larger than 100 MB ca. due to its low memory availability. Turnaround time and peak memory results shown in Table 1 and Figs. 3c and 3d follow the hardware availability, *i.e.*, devices with faster CPUs have lower turnaround times and those with larger RAM have been granted more memory by the OS. An interesting result is that the Mac Pro, plotted alongside the mobile devices in Fig. 3c, has been quicker than the iPhone 7 when reasoning over smaller ontologies, but it has been clearly outperformed

starting from medium-sized ones. This is likely due to the implemented algorithms being single-threaded, not exploiting the higher level of parallelism in the desktop CPU w.r.t. mobile ones. Furthermore, Fig. 3d shows lower memory peaks for small ontologies on the Mac Pro: since all iOS applications must have a graphical user interface, mobile tests are affected by a systematic memory overhead w.r.t. tests executed through command line interface on the Mac[17]. In order to provide further insight, Fig. 4 compares all reasoners (on workstation and mobile) on the whole set of KBs which have been correctly processed by all tools. The area with blue background encloses Mini-ME Swift instances running on desktop and mobile devices. In Fig. 4a inference tasks are split in ontology parsing and actual reasoning; the overall time spent in each step is displayed on the axes. Figure 4b shows on both the axes the lowest and highest memory peak values across the whole dataset. Although iPhone 5s maximum peak value is biased by failed tests, the overall results highlight Mini-ME Swift has a comparatively low memory footprint.

Table 2. Features of KBs used for non-standard tests and memory peak (MB)

		Toy	Agriculture	Building	MatchAndDate
Ontology features	Size (kB)	30.43	128.35	142.08	590.54
	#concepts	48	134	180	157
	#roles	8	17	27	11
	#instances	7	16	29	100
	#matchmaking	28	48	493	10000
Memory peak (MB)	Mac Pro (Java)	53.48	68.72	75.46	130.40
	Mac Pro (Swift)	6.26	6.78	6.98	8.55
	iPhone 7	15.76	16.09	18.09	19.22
	iPad Mini 4	14.72	14.22	14.36	16.16
	iPhone 5s	9.60	9.97	10.05	12.37

5.2 Non-standard Inference Services

Non-standard inference services have been evaluated on four $\mathcal{ALN}$ KBs. Their features are summarized in the upper section of Table 2: size, number of concepts, roles, instances and number of matchmaking tasks to be performed in the tests, where each task has been executed on a $\langle$*request, resource*$\rangle$ pair involving the following steps: (i) resource and request are checked for compatibility; (ii) if they are compatible, CA is performed; otherwise CC is executed, followed by CA on the compatible part of the request [26]. Mini-ME Java has been used as

[17] Mobile tests could have been run from command line interface by *jailbreaking* iOS devices. This has been avoided, in order to assess performance in the standard iOS configuration.

test oracle and reference for performance comparison, because other reasoners evaluated in Sect. 5.1 do not provide Concept Abduction and Contraction.

Correctness. Mini-ME Swift has correctly performed the CA and CC tests for all the pairs of individuals in the test set.

Performance. As shown in Fig. 5 (average time per matchmaking task) and the lower section of Table 2 (memory peak), Mini-ME Swift significantly outperforms its Java counterpart on the desktop. Memory usage is an order of magnitude lower, also due to the overhead induced by the Java Virtual Machine.

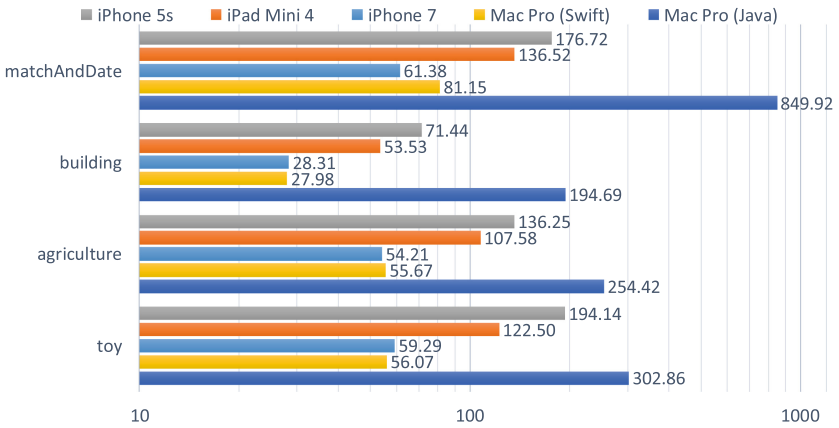


Fig. 5. Average time per matchmaking task (μs)

6 Conclusion and Future Work

The paper introduced Mini-ME Swift, the first OWL reasoner for iOS. It runs natively on iOS for SWoT scenarios, as well as on macOS for classical Semantic Web use cases, providing standard and non-standard inference services. Several optimization techniques have allowed for satisfactory time and memory performance, as evidenced in a comparative assessment with popular Semantic Web reasoners on desktop and mobile testbeds.

Future work concerns the extension of the supported logic language with concrete domains and $\mathcal{EL}$ (Existential Languages) family. Furthermore, implementing an OWL functional syntax parser will enable performance improvements. Concerning the prototype in Sect. 4, early investigations are ongoing on the integration of semantic-enhanced navigation by embedding inferences within the *GraphHopper* routing engine for iOS[18].

[18] GraphHopper iOS port: https://github.com/graphhopper/graphhopper-ios.

References

1. Ali, S., Kiefer, S.: μOR – a micro OWL DL reasoner for ambient intelligent devices. In: Abdennadher, N., Petcu, D. (eds.) GPC 2009. LNCS, vol. 5529, pp. 305–316. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01671-4_28
2. Baader, F., Hollunder, B., Nebel, B., Profitlich, H.J., Franconi, E.: Am empirical analysis of optimization techniques for terminological representation systems. Appl. Intell. **4**(2), 109–132 (1994)
3. Bernstein, A., Hendler, J., Noy, N.: A new look at the Semantic Web. Commun. ACM **59**(9), 35–37 (2016)
4. Bobed, C., Yus, R., Bobillo, F., Mena, E.: Semantic reasoning on mobile devices: do Androids dream of efficient reasoners? J. Web Semant. **35**, 167–183 (2015)
5. Ermilov, T., Khalili, A., Auer, S.: Ubiquitous semantic applications: a systematic literature review. Int. J. Semant. Web Inf. Syst. **10**(1), 66–99 (2014)
6. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z.: HermiT: an OWL 2 reasoner. J. Autom. Reasoning **53**(3), 245–269 (2014)
7. Horridge, M., Bechhofer, S.: The OWL API: a java API for OWL ontologies. Semant. Web **2**(1), 11–21 (2011)
8. Jara, A.J., Olivieri, A.C., Bocchi, Y., Jung, M., Kastner, W., Skarmeta, A.F.: Semantic Web of Things: an analysis of the application semantics for the IoT moving towards the IoT convergence. Int. J. Web Grid Serv. **10**(2–3), 244–272 (2014)
9. Kazakov, Y., Klinov, P.: Experimenting with ELK reasoner on android. In: OWL Reasoner Evaluation Workshop (ORE). CEUR Workshop Proceedings, vol. 1015, pp. 68–74. CEUR-WS, Aachen (2013)
10. Kim, T., Park, I., Hyun, S.J., Lee, D.: MiRE4OWL: mobile rule engine for OWL. In: Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops, pp. 317–322. IEEE Computer Society, Piscataway (2010)
11. Koch, F., Meyer, J.-J.C., Dignum, F., Rahwan, I.: Programming deliberative agents for mobile services: the 3APL-M platform. In: Bordini, R.H., Dastani, M.M., Dix, J., El Fallah Seghrouchni, A. (eds.) ProMAS 2005. LNCS (LNAI), vol. 3862, pp. 222–235. Springer, Heidelberg (2006). https://doi.org/10.1007/11678823_14
12. Koziuk, M., Domaszewicz, J., Schoeneich, R.O., Jablonowski, M., Boetzel, P.: Mobile context-addressable messaging with DL-lite domain model. In: Roggen, D., Lombriser, C., Tröster, G., Kortuem, G., Havinga, P. (eds.) EuroSSC 2008. LNCS, vol. 5279, pp. 168–181. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88793-5_13
13. Li, Y.F., Pan, J.Z., Hauswirth, M., Nguyen, H.: The ubiquitous Semantic Web: promises, progress and challenges. In: Web-Based Services: Concepts, Methodologies, Tools, and Applications, pp. 272–289. IGI Global, Hershey (2016)
14. Matentzoglu, N., Leo, J., Hudhra, V., Sattler, U., Parsia, B.: A survey of current, stand-alone OWL reasoners. In: 4th OWL Reasoner Evaluation Workshop (ORE). CEUR Workshop Proceedings, vol. 1387, pp. 68–79. CEUR-WS, Aachen (2015)
15. McBride, B.: Jena: a Semantic Web toolkit. IEEE Internet Comput. **6**(6), 55–59 (2002)
16. Moguillansky, M.O., Wassermann, R., Falappa, M.A.: An argumentation machinery to reason over inconsistent ontologies. In: Kuri-Morales, A., Simari, G.R. (eds.) IBERAMIA 2010. LNCS (LNAI), vol. 6433, pp. 100–109. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16952-6_11

17. Motik, B., Horrocks, I., Kim, S.M.: Delta-reasoner: a Semantic Web reasoner for an intelligent mobile platform. In: Proceedings of the 21st International Conference on World Wide Web, pp. 63–72. ACM, New York (2012)
18. Parsia, B., Matentzoglu, N., Gonçalves, R.S., Glimm, B., Steigmiller, A.: The OWL reasoner evaluation (ORE) 2015 competition report. J. Autom. Reasoning **59**(4), 455–482 (2017)
19. Patton, E.W., McGuinness, D.L.: The mobile wine agent: pairing wine with the social Semantic Web. In: 2nd Social Data on the Web Workshop - 8th International Semantic Web Conference. CEUR Workshop Proceedings, vol. 520. CEUR-WS, Aachen (2009)
20. Pizzocaro, D., Preece, A., Chen, F., La Porta, T., Bar-Noy, A.: A distributed architecture for heterogeneous multi sensor-task allocation. In: 2011 International Conference on Distributed Computing in Sensor Systems (DCOSS), pp. 1–8. IEEE, Piscataway (2011)
21. Ruta, M., Scioscia, F., Di Sciascio, E.: Mobile semantic-based matchmaking: a fuzzy DL approach. In: Aroyo, L., et al. (eds.) ESWC 2010. LNCS, vol. 6088, pp. 16–30. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13486-9_2
22. Ruta, M., Scioscia, F., Di Sciascio, E., Bilenchi, I.: OWL API for iOS: early implementation and results. In: Dragoni, M., Poveda-Villalón, M., Jimenez-Ruiz, E. (eds.) OWLED/ORE -2016. LNCS, vol. 10161, pp. 141–152. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-54627-8_11
23. Ruta, M., Scioscia, F., Gramegna, F., Di Sciascio, E.: A mobile knowledge-based system for on-board diagnostics and car driving assistance. In: 4th International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM), pp. 91–96. ThinkMind, Wilmington (2010)
24. Ruta, M., Scioscia, F., Ieva, S., De Filippis, D., Di Sciascio, E.: Indoor/outdoor mobile navigation via knowledge-based POI discovery in augmented reality. In: Web Intelligence and Intelligent Agent Technology (WI-IAT), vol. 3, pp. 26–30. IEEE, Piscataway (2015)
25. Scioscia, F., Ruta, M.: Building a Semantic Web of Things: issues and perspectives in information compression. In: Proceedings of the 3rd IEEE International Conference on Semantic Computing, pp. 589–594. IEEE Computer Society, Piscataway (2009)
26. Scioscia, F., Ruta, M., Loseto, G., Gramegna, F., Ieva, S., Pinto, A., Di Sciascio, E.: Mini-ME matchmaker and reasoner for the Semantic Web of Things. In: Innovations, Developments, and Applications of Semantic Web and Information Systems, pp. 262–294. IGI Global, Hershey (2018)
27. Seitz, C., Schönfelder, R.: Rule-based OWL reasoning for specific embedded devices. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011. LNCS, vol. 7032, pp. 237–252. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25093-4_16
28. Sinner, A., Kleemann, T.: KRHyper – in your pocket. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 452–457. Springer, Heidelberg (2005). https://doi.org/10.1007/11532231_33
29. Steigmiller, A., Liebig, T., Glimm, B.: Konclude: system description. J. Web Semant. **27**, 78–85 (2014)
30. Steller, L., Krishnaswamy, S.: Pervasive service discovery: mTableaux mobile reasoning. In: International Conference on Semantic Systems (I-Semantics), pp. 93–101. TU Graz, Graz (2008)
31. Tai, W., Keeney, J., O'Sullivan, D.: Resource-constrained reasoning using a reasoner composition approach. Semant. Web **6**(1), 35–59 (2015)

32. Teege, G.: Making the difference: a subtraction operation for description logics. In: Proceedings of the Fourth International Conference on the Principles of Knowledge Representation and Reasoning (KR 1994), pp. 540–550. ACM, New York (1994)
33. Thomas, E., Pan, J.Z., Ren, Y.: TrOWL: tractable OWL 2 reasoning infrastructure. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) ESWC 2010. LNCS, vol. 6089, pp. 431–435. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13489-0_38
34. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: system description. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 292–297. Springer, Heidelberg (2006). https://doi.org/10.1007/11814771_26
35. van Aart, C., Wielinga, B., van Hage, W.R.: Mobile cultural heritage guide: location-aware semantic search. In: Cimiano, P., Pinto, H.S. (eds.) EKAW 2010. LNCS (LNAI), vol. 6317, pp. 257–271. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16438-5_18
36. Van Woensel, W., Abidi, S.S.R.: Optimizing semantic reasoning on memory-constrained platforms using the RETE algorithm. In: Gangemi, A., Navigli, R., Vidal, M.-E., Hitzler, P., Troncy, R., Hollink, L., Tordai, A., Alam, M. (eds.) ESWC 2018. LNCS, vol. 10843, pp. 682–696. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93417-4_44
37. Yus, R., Pappachan, P.: Are apps going semantic? a systematic review of semantic mobile applications. In: 1st International Workshop on Mobile Deployment of Semantic Technologies (MoDeST). CEUR Workshop Proceedings, vol. 1506, pp. 2–13. CEUR-WS, Aachen (2015)