Web Semantics: Science, Services and Agents on the World Wide Web 73 (2022) 100709

Contents lists available at ScienceDirect

Web Semantics: Science, Services and Agents on the World Wide Web

journal homepage: www.elsevier.com/locate/websem

# A multiplatform reasoning engine for the Semantic Web of Everything

Michele Ruta *, Floriano Scioscia, Ivano Bilenchi, Filippo Gramegna, Giuseppe Loseto, Saverio Ieva, Agnese Pinto

*Polytechnic University of Bari, Department of Electrical and Information Engineering, via E. Orabona 4, I-70125, Bari, Italy*

ARTICLE INFO

*Article history:*
Received 21 January 2021
Received in revised form 23 January 2022
Accepted 25 February 2022
Available online 8 March 2022

*Keywords:*
Automated reasoning
Web Ontology Language
Internet of Everything
Semantic Web of Things
Software architecture

ABSTRACT

The Internet of Everything and Semantic Web can be joined by giving more intelligence to pervasive systems. To that end, reasoning capabilities should be enabled even for very resource-constrained embedded devices. This paper presents *Tiny-ME* (the Tiny Matchmaking Engine), a matchmaking and reasoning engine for the Web Ontology Language (OWL), designed and implemented with a compact and portable C core. Main features are high resource efficiency and multiplatform support, spanning containerized microservices, desktops, mobile devices, and embedded boards. The OWLlink interface has been extended to enable non-standard reasoning services for matchmaking in Web, Cloud, and Edge computing. A prototype evaluation is proposed, including a case study on the *Pixhawk* Unmanned Aerial Vehicle (UAV) autopilot and performance highlights.



"*It is the wise who prevail in every field*". [1]

## 1. Introduction

Objects and environments – both natural and man-made – are increasingly augmented with networking capabilities with the Internet of Things (IoT), including hundreds of millions of people using wearable devices to gather sensitive health data continuously. The interconnection of people, things, processes, and data is shaping the so-called *Internet of Everything* (IoE), creating new opportunities in a wide range of economic and societal domains [2]. Information processing and management requires intelligence at all scales, from nano-networks to the World Wide Web. With such diversification, device and platform heterogeneity exacerbates IoT interoperability issues.

The Semantic Web of Things (SWoT) [3] has promoted the integration of Semantic Web languages and technologies in IoT contexts, in order to imbue environments with knowledge-based capabilities. In the SWoT vision, ontology-based annotations are associated to devices, objects, and phenomena to describe them in a rich and structured way, with unambiguous semantics supporting automated reasoning procedures to infer implicit knowledge. The progression of the IoT toward the IoE calls for a corresponding evolution of the SWoT toward a *Semantic Web of Everything* (SWoE), where semantic technologies permeate and enable interactions among people, things, processes, and data, from WWW- to micro- and nano-scale. This vision requires supporting Knowledge Representation (KR) languages and automated inference on tiny autonomous devices with very strict processing, memory, and energy constraints. Inferring implicit knowledge via automated reasoning in Internet of Everything scenarios is useful to grant (groups of) tiny devices with autonomous decision making, knowledge-based self-coordination and self-management, as well as timely and unobtrusive decision support to end users. In the SWoE, inference procedures must be available locally, since the reachability of more powerful companion devices acting as semantic facilitators cannot be taken for granted. High volatility of devices, resources and data requires quick on-the-fly processing capabilities, which are not always available on external devices or cannot tolerate the latencies of wireless low-power network links in request–response interactions. After all, *fog computing* has shown that a capillary distribution of data processing can improve efficiency, timeliness, and security of large-scale data collection and analysis [4]. Reasoning engine designs are currently oriented to the WWW or mobile devices such as tablets and smartphones. Unfortunately, this is not adequate for the SWoE, where target platforms include tiny inexpensive sensors and boards like those embedded in wearable devices or shopfloor and medical equipment.

This paper introduces *Tiny-ME*[1] (the Tiny Matchmaking Engine), a SWoE-oriented matchmaking and reasoning engine. It exhibits a multiplatform architecture from the ground up, with

* Corresponding author.
*E-mail addresses:* michele.ruta@poliba.it (M. Ruta), floriano.scioscia@poliba.it (F. Scioscia), ivano.bilenchi@poliba.it (I. Bilenchi), filippo.gramegna@poliba.it (F. Gramegna), giuseppe.loseto@poliba.it (G. Loseto), saverio.ieva@poliba.it (S. Ieva), agnese.pinto@poliba.it (A. Pinto).

https://doi.org/10.1016/j.websem.2022.100709
1570-8268/

[1] Tiny-ME home: http://swot.sisinflab.poliba.it/tinyme/.

a common core in C language granting both portability and efficient implementation of Knowledge Base (KB) management and reasoning services. On top of this low-level C API (Application Programming Interface), three high-level APIs for working with OWL 2 (Web Ontology Language) [5] knowledge bases are provided: (i) the *OWL API* [6] for Java-based platforms; (ii) the *OWL API for iOS*, [7] for Objective-C and Swift development; (iii) *Cowl*, a novel lightweight C API for native C/C++ development in UNIX-like, Windows, and embedded environments.

The reasoner provides standard (Ontology Classification, Ontology Coherence, Concept Subsumption, and Concept Satisfiability checks) and non-standard (Concept Abduction, Contraction, Bonus, Difference, and Covering) inference services on moderately expressive KBs, in an OWL 2 subset corresponding to the *Attributive Language with unqualified Number restrictions* ($\mathcal{ALN}$) Description Logic (DL). This choice was made because standard and non-standard inference algorithms on concept expressions can be carried out using structural algorithms in polynomial time and space [8], a property that is particularly well-suited for heavily resource-constrained devices.

All major desktop and mobile operating systems (OSes) are compatible out of the box, which implies that Tiny-ME also runs in *Docker*[2] containers, enabling reasoning in microservice architectures for Web and Cloud applications: to this aim, the standard *OWLlink* [9] protocol for remote reasoner invocation has been extended, adding support for the provided non-standard inference services. Furthermore, Tiny-ME is conceived to be easily compatible with most OSes and platforms with basic C support: in order to validate this claim, as a case study and illustrative example, it has been ported to the *Apache NuttX*[3] real-time operating system (RTOS) for the *Pixhawk*[4] open standard platform for Unmanned Aerial Vehicle (UAV) autopilots. The NuttX/Pixhawk platform has been chosen because it exhibits the typical development challenges of embedded RTOSs, while also introducing very strict constraints on computational resources. A preliminary performance evaluation has been then carried out with respect to the *Konclude* [10], *Mini-ME 2.0* [8] and *Mini-ME Swift* [11] reasoners on both desktop and mobile devices.

The core contributions of this paper can be summarized as follows:

- the Tiny-ME SWoE-oriented reasoning and matchmaking engine, supporting a set of standard and non-standard inference services in a moderately expressive OWL 2 fragment corresponding to the $\mathcal{ALN}$ DL;
- a new multiplatform architecture, natively supporting Windows, Linux, macOS, Docker containers, Android, and iOS, further minimizing the porting effort to other platforms;
- re-engineered inference procedures optimized for SWoT contexts;
- Cowl, a new C parser for OWL 2 Full, embedded in Tiny-ME but also released independently with a permissive open source license (Section 3.4) to promote reuse;
- extension of the OWLlink interface for HTTP-based application-reasoner client–server interaction, supporting non-standard inference services for semantic matchmaking;
- a case study on the Pixhawk UAV autopilot demonstrating (i) the ease of porting to the Apache NuttX RTOS and (ii) the feasibility of implementing a simple but useful application with a satisfactory expressiveness given the hardware constraints of the target embedded devices.

Together, these contributions provide a new worthwhile platform for the realization of successful ontology-based systems in SWoT/SWoE contexts. Moreover, selected contributions like Cowl, the OWLlink extension, and the multiplatform architectural model itself can be mutuated by other ontology-based software systems for further purposes.

The remainder of this paper is organized as follows: after related work in Sections 2 and 3 discusses Tiny-ME's inference services and architecture in detail, including the proposed OWLlink extension. The Pixhawk case study follows in Section 4, before performance results in Section 5 and conclusion.

## 2. Related work

Classical Semantic Web contexts are characterized by stable availability of hefty computational and networking resources. Conversely, in SWoE scenarios hardware is severely constrained and information sources are micro-devices scattered throughout physical environments, so that their availability is unpredictably intermittent due to user and device mobility as well as limitations of wireless communication links and energy supply. For this reason, batch workloads on very expressive and complex Knowledge Bases are out of place in the SWoE, while inference engines should support quick query processing on relatively small and simple annotations. These requirements have a direct impact on the expressiveness of the logical languages to be managed. Each language in the DL family is distinguished by its set of available constructs. Including more constructs grants larger expressiveness, but also leads to an (oftentimes steep) increase of computational complexity of inferences [12]. A trade-off is therefore worthwhile. Historically, the untractable worst-case complexity of OWL DL stimulated the characterization of OWL 2 *profiles*, which simplified the languages and admitted KB axioms, so that efficient algorithms could be adopted while keeping enough expressiveness for practically relevant use cases. $\mathcal{EL}^{++}$ [13] extended the basic $\mathcal{EL}$ DL to suit various applications, characterized by very large ontologies with moderate expressiveness. The proposal included a polynomial-complexity structural Ontology Classification algorithm, which enabled the development of high-performance $\mathcal{EL}^{++}$ classifiers such as ELK [14]. Analogously, one of the earliest approaches to adapt Concept Contraction and Concept Abduction non-standard inferences to pervasive computing [15] adopted structural algorithms on acyclic TBoxes in the $\mathcal{AL}$ language, implementing the above algorithms through a mobile Relational DataBase Management System (RDBMS) query layer.

The earliest generation of mobile and IoT-oriented inference engines adopted simplified approaches with respect to existing Semantic Web reasoners to cope with the limited memory availability of devices. *Pocket KRHyper* [16], a Java Micro Edition library for theorem proving and model generation based on the hyper tableau calculus, was the earliest example of lightweight inference engine, albeit suffering from the severe memory limitations of the target platform. The *μOR* [17] reasoner implemented a simple resolution and pattern matching algorithm on a subset of OWL-Lite. Analogously, *MiRE4OWL* [18] was a rule-based mobile inference engine leveraging OWL-DL semantics, resolved with the classic RETE algorithm.

More recently, efforts were devoted to miniaturize reasoning engines for embedded devices. In [19] consequence-driven $\mathcal{EL}^{+}$ reasoning was ported to a Programmable Logic Controller (PLC) for industrial automation. The modular rule-based reasoner in [20] combined selective rule loading and a two-stage RETE algorithm; it exhibited satisfactory performance on the *Sun SPOT* sensor platform for small- and medium-sized ontologies.

A second group of reasoners is the one ported from conventional computer platforms to mobile devices, leveraging the

[2] Docker home: https://www.docker.com/.
[3] Apache NuttX home: https://nuttx.apache.org/.
[4] Pixhawk home: https://pixhawk.org/.

increasing availability of computational resources. In [21], five Java OWL reasoners have been adapted for Android, albeit with not-negligible effort. In [22] ELK has been re-engineered from Java Standard Edition (SE) to Android, minimizing memory consumption and implementing support for multi-core CPUs. Native mobile inference engines also exist. *Mini-ME 2.0* [8] is an Android-oriented *matchmaker* and reasoner compatible with Java SE.

Semantic matchmaking is defined as the problem of finding the most relevant element in a set of *resources* w.r.t. a given *request*, where both request and resources are represented as satisfiable concept expressions w.r.t. a common set of axioms $\mathcal{T}$ in a DL. The output of semantic matchmaking consists of a conjunction of concepts, each with a score representing its semantic relevance w.r.t. the submitted query. Besides resource discovery, matchmaking can support data stream analytics, *e.g.*, statistical classification problems could be converted to semantic matchmaking problems. As an example, in [23] features of samples were annotated with conjunctive concept expression fragments and target classes with concept expressions, both referring to a common ontology. These two use cases are highly relevant in SWoT/SWoE scenarios, characterized by volatile data and fragmented information [24–26]. Tiny-ME refers to the theoretical framework grounded on the classification of match types proposed in [27]. Similar classifications were suggested in [28, 29]. Differing from those earlier works, however, the framework adopted by Tiny-ME has a different order of preference among the various non-exact match categories, with *full/subsume* being the favorite one due to not needing to hypothesize additional information about the resource, as explained in Section 3.2. Furthermore, the adopted framework enables ranking different resources within the same match category. More recent works [30, 31] are oriented to resource discovery in the SWoT and combine reasoning and quantitative contextual attributes in *utility functions*.

[32] presents a rule engine for Android introducing the novel *RETEpool* algorithm on OWL 2 RL rulesets, capable of balancing memory usage and time performance. *Mini-ME Swift* [11] is the first OWL reasoner for iOS, re-designed from the above Mini-ME with the OWL API for iOS [7]. Tiny-ME expands on ideas behind Mini-ME and Mini-ME Swift by introducing multiple novelties: a completely new multiplatform architecture (Section 3.2); re-engineered and optimized inference procedures (Section 3.1); a new OWL 2 parser in C, named Cowl, released with a permissive open source license (Section 3.4); a new OWLlink interface extending the specification for non-standard inference services (Section 3.3).

Besides Java-based OWL tools, cross-platform reasoners projects include the mobile semantic rule engine evaluation framework in [33], developed in JavaScript with the *PhoneGap*[5] Software Development Kit (SDK), and its successor *MobiBench* [34]. The former supported integration with rule engines written either in JavaScript or natively for one of the PhoneGap target platforms (Android, iOS, Windows 8.1), the latter also supports Java engines through the *Nashorn* JavaScript interpreter included in Java SE (version 8 and later). The *Owlready2* library [35] implements OWL KB manipulation and object-oriented mappings in Python, adopting *HermiT* [36] as backend for inferences.

Concerning automated reasoning at a Web scale, algorithm optimization for distributed execution in Cloud environments has been first proposed by *WebPIE* [37]. Tiny-ME aims to support cloud environments through an extension of the OWLlink protocol [9] for client–server interaction: the standard specification is supported by *Konclude* [10] and many more Semantic Web reasoners.

Table 1 summarizes relevant features of related reasoning systems, highlighting Tiny-ME's contributions. Some columns require clarification: the "*Full*" label refers to systems supporting only *exact* or full/subsume match degrees [28], thus requiring that resources are subsumed by request; the "*Approximated*" label marks systems that are able to support further match degrees like *potential/intersection* and *partial/disjoint*, respectively when subsumption does not hold and when the conjunction of request and resource is unsatisfiable. Finally, the *Explanation* column refers to the ability of systems to provide formal justifications for their outcomes, *e.g.*, "why" subsumption does not hold between two concept expressions.

## 3. Reasoning in the Semantic Web of Everything

When designing a reasoner which can seamlessly adapt to a wide range of use cases, difficulties arise from the need for it to be simultaneously flexible, easy to use and maintain, and highly resource-efficient. Neglecting even just one of these requirements leads to making the applicability to SWoE use cases unlikely. In order to meet the above goals, the design of Tiny-ME has followed a few fundamental criteria:

- Inference services should be implemented only once and should work on the widest possible platform range. This is because reasoning algorithm porting is hard to carry out, particularly when optimization comes into play: shared implementations significantly improve the overall system maintainability.
- The implementation of inference services must be very efficient, especially for what concerns memory usage. This really enables deployment on tiny resource-constrained devices.
- Applications should have simple APIs to use the reasoner. This leads to the exposure of APIs in multiple target languages. If an API is not available for a given language, it should be possible to create it with reasonable effort, based on the existing implementation of inference algorithms. This ensures the system is flexible enough and easy to use.
- Knowledge representation and reasoning functionalities should be separated in distinct modules. KR comes at a significant cost in terms of computational resources, as OWL data models and parsers tend to be rather large, both code- and data-wise. In common interchange syntaxes for the Semantic Web, language constructs are basically strings, which take up significant amounts of memory and sometimes are not strictly needed in reasoning procedures. Furthermore, the OWL 2 specification mandates support for the RDF/XML [38] serialization, which implies the presence of XML parsers, further increasing code size and memory demand.

The subsections hereafter detail the overall reasoner architecture, focusing on the available inference services, the proposed OWLlink extension, and the platform-specific APIs.

### 3.1. Language and inference services

The proposed system implements polynomial-complexity structural algorithms on $\mathcal{ALN}$ concept expressions (see Table 2 for a list of supported constructs), based on the concept *unfolding* and *Conjunctive Normal Form* (CNF) normalization preprocessing steps [8]. Basically, concept unfolding recursively expands terminological axioms in the ontology (a.k.a. TBox, Terminological

[5] PhoneGap home: https://phonegap.com/.

**Table 1**
Features of related reasoning systems.

| Name | Features | | | | Platforms | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | DL | Algorithm family | Matchmaking | Explanation | Language | Web/Cloud | Desktop | Mobile | Embedded |
| $\mu$OR [17] | OWL-Lite$^-$ | Pattern matching | Full | | Java | | JVM | | JamVM |
| COROR [20] | $\mathcal{SHOIN}$(D) | RETE | Full | | Java | | JVM | | Sun SPOT |
| ELK [22] | $\mathcal{EL}^+$ | Consequence based | | | Java | | JVM | Android | |
| HermiT [21,36] | $\mathcal{SROIQ}$(D) | Tableaux | Full | | Java | OWLlink API | JVM | Android | |
| JFact [21] | $\mathcal{SROIQ}$(D) | Tableaux | Full | | Java | OWLlink API | JVM | Android | |
| Konclude [10] | $\mathcal{SROIQV}$(D) | Hybrid | Full | | C++ | OWLlink built-in | Native | | |
| Mini-ME [8] | $\mathcal{ALN}$ | Structural | Full, Approximated | Yes | Java | OWLlink API | JVM | Android | |
| Mini-ME Swift [11] | $\mathcal{ALN}$ | Structural | Full, Approximated | Yes | Swift | | macOS | iOS | |
| MiRE4OWL [18] | $\mathcal{SHOIN}$(D) | RETE | Full | | C++ | | Native | Windows Mobile | |
| Pellet [21] | $\mathcal{SROIQ}$(D) | Hybrid | Full | Yes | Java | | JVM | Android | |
| PLC-based [19] | $\mathcal{EL}^+$ | Consequence based | | | SCL | | | | Siemens SIMATIC |
| Pocket KRHyper [16] | $\mathcal{ALC}^+$ | Tableaux | Full | | Java | | JVM | J2ME | |
| RETEpool [32] | OWL-RL | RETE | Full | | Java | | JVM | Android | |
| Tiny-ME | $\mathcal{ALN}$ | Structural | Full, Approximated | Yes | C, Obj-C, Java | OWLlink API | Native, JVM | Android, iOS | Any with C support |

**Table 2**
$\mathcal{ALN}$ constructs supported by Tiny-ME.

| Construct | Syntax |
|---|---|
| Top | $\top$ |
| Bottom | $\bot$ |
| Intersection | $C \sqcap D$ |
| Atomic negation | $\neg A$ |
| Universal quantification | $\forall R.C$ |
| Number restrictions | $\geq nR$<br>$\leq nR$ |
| Atomic inclusion | $A \sqsubseteq C$ |
| Atomic equivalence | $A \equiv C$ |

Box) within concept expressions, so that the TBox is not needed anymore during subsequent inferences.

In order to have finite unfoldings, and to ensure polynomial time and space complexity of inference procedures, TBoxes are subject to the following limitations [30]:

- they must be acyclic, though Tiny-ME partially relaxes this requirement by admitting *told subsumption cycles* [39];
- the left-hand side (LHS) of inclusion ($\sqsubseteq$) and definition ($\equiv$) axioms must be atomic, *i.e.*, *general concept inclusions* are not currently supported;
- if an atomic concept $A$ is the LHS of a definition axiom, then it cannot be the LHS of any other inclusion or definition axioms.

CNF normalization translates the unfolded concept expression in a canonical form that preserves its semantics. 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 fillers recursively in CNF.

Structural algorithms for standard reasoning tasks are well known (see [12], 2.3.1). In $\mathcal{ALN}$ DL, once concept expressions have been unfolded and normalized, comparisons between them basically come down to set operations. As an example, let us consider the following $\mathcal{ALN}$ TBox (named $TB_{ex}$):

$$A \equiv \forall P.D \sqcap (\geq 3P)$$
$$B \sqsubseteq \forall P.D$$
$$C \sqsubseteq B \sqcap (\geq 2P)$$

By applying unfolding and CNF normalization, we get the following concept expressions (CEs):

$$A \rightarrow \forall P.D \sqcap (\geq 3P)$$
$$B \rightarrow B \sqcap \forall P.D$$
$$C \rightarrow B \sqcap C \sqcap \forall P.D \sqcap (\geq 2P)$$

It can be noticed $B$ and $C$ appear among the conjuncts of their own unfolded concept expressions, while $A$ does not: as explained in [12, §9.2.3], in $\mathcal{ALN}$ DL an atomic concept must be included in its own unfolding iff it is the LHS of an inclusion axiom, while it is omitted if it is the LHS of an equivalence one.

Let us suppose we need to check whether $A \sqsubseteq C$ holds: after unfolding and normalizing, we just need the CEs of $A$ and $C$, *i.e.*, the whole TBox is not required anymore. In this case, $A \sqsubseteq C$ holds because $\forall P.D$ is in both the CEs of $A$ and $C$, and $(\geq 3P)$ is more specific than $(\geq 2P)$.

Tiny-ME supplies *standard* Subsumption and Satisfiability checks over concept expressions. **Satisfiability** is trivially checked by performing CNF normalization [8]: as previously stated, a CNF-normalized concept expression $A$ is either $\bot$, or the conjunction

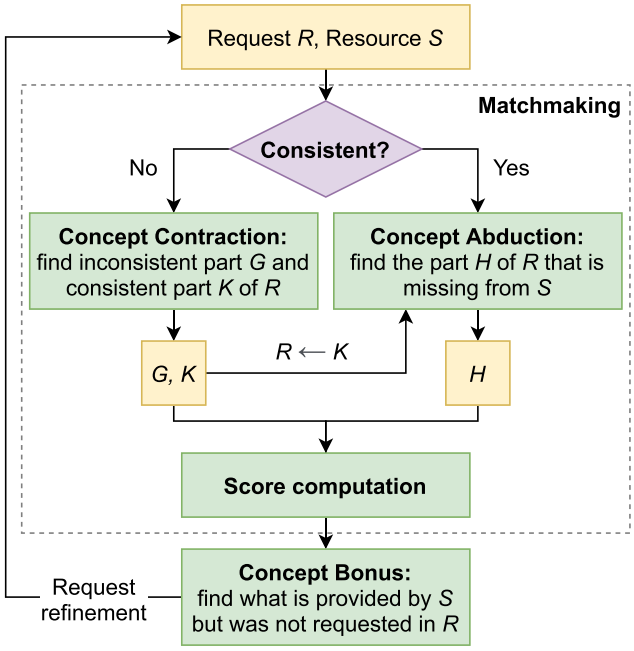


**Fig. 1.** Semantic matchmaking framework.

of an arbitrary number of supported constructs. In order to check whether $A$ is satisfiable, it is sufficient to verify that it is not $\bot$. **Subsumption** exploits the classic structural algorithm in [12]. Given two CNF-normalized concept expressions $R$ and $S$, to assess if $R \sqsubseteq S$ holds:

1. if $R \equiv \bot$, then $R \sqsubseteq S$.
2. for each atomic concept $A$ in $S$, if $A$ is not in $R$, then $R \not\sqsubseteq S$.
3. for each negated atomic concept $\neg A$ in $S$, if $\neg A$ is not in $R$, then $R \not\sqsubseteq S$.
4. for each role $P$ such that $\leq xP$ is in $S$, if $\leq yP$ with $x < y$ is in $R$, then $R \not\sqsubseteq S$.
5. for each role $P$ such that $\geq xP$ is in $S$, if $\geq yP$ with $x > y$ is in $R$, then $R \not\sqsubseteq S$.
6. for each role $P$ such that $\forall P.E$ is in $S$, if $\forall P.F$ with $F \not\sqsubseteq E$ is in $R$, then $R \not\sqsubseteq S$.
7. otherwise, $R \sqsubseteq S$.

Standard inference services are not enough in scenarios requiring more than just a Boolean answer, such as in matchmaking or negotiation. In those cases, *non-standard* Concept Abduction and Concept Contraction [30] are more useful, as they provide justifications for (missed) subsumption and (un)satisfiability, in the Open World Assumption. Tiny-ME supports the *semantic matchmaking* framework sketched in the flow chart of Fig. 1 and outlined hereafter.

Let us consider a set of axioms $\mathcal{T}$ in $\mathcal{ALN}$, and $R$, $S$ two concepts in $\mathcal{ALN}$ – representing a *request* and a *resource*, respectively – both satisfiable in $\mathcal{T}$. A preliminary **Consistency check** is performed to assess whether $R \sqcap S$ is satisfiable w.r.t. $\mathcal{T}$; in formulae, $\mathcal{T} \models R \sqcap S \not\sqsubseteq \bot$. If the check fails the resource is a *partial match* for the request, and **Concept Contraction** (CC) can be computed. Its output consists of a pair of concepts $\langle G, K \rangle$ such that $\mathcal{T} \models R \equiv G \sqcap K$, and $\mathcal{T} \models K \sqcap S \not\sqsubseteq \bot$. Basically, Contraction determines which part of $R$ clashes with $S$. By retracting only conflicting requirements $G$ (for *Give up*) from $R$, an expression $K$ (for *Keep*) remains, *i.e.*, a contracted version of the original request. The solution $G$ to Contraction explains "why" the conjunction of $R$ and $S$ is not satisfiable, providing a way to move from a partial match to a *potential match* situation. Having $R$ and $S$ unfolded and CNF-normalized concept expressions in $\mathcal{ALN}$, $CC(R, S)$ is computed by means of the following structural algorithm:

1. set $K := R$ and $G := \top$;
2. for each atomic concept $A$ in $K$, if $\neg A$ is in $S$, then move $A$ from $K$ to $G$;
3. for each negated atomic concept $\neg A$ in $K$, if $A$ is in $S$, then move $\neg A$ from $K$ to $G$;
4. for each role (object property) $P$ such that $\geq xP$ is in $K$ and $\leq yP$ is in $S$ with $y < x$, replace $\geq xP$ in $K$ with $\geq yP$ and put $\geq xP$ in conjunction with the concept expression for $G$;
5. for each role $P$ such that $\leq xP$ is in $K$ and $\geq yP$ is in $S$ with $y > x$, replace $\leq xP$ in $K$ with $\leq yP$ and put $\leq xP$ in conjunction with the concept expression for $G$;
6. for each role $P$ s.t. $\forall P.E$ is in $K$ and $\forall P.F$ is in $S$, if $\exists \geq xP$ with $x > 0$ is either in $K$ or in $S$, then compute Contraction recursively on the fillers: $\langle G', K' \rangle = CC(E, F)$, put $\forall P.G'$ in conjunction with the concept expression for $G$ and replace $\forall P.E$ with $\forall P.K'$ in the concept expression for $K$.

For example, referring to the above $TB_{ex}$, $\langle G, K \rangle = CC(A, \leq 2\ P) = \langle \geq 3\ P, \geq 2\ P \rangle$.

**Concept Abduction** (CA) can be computed in case of potential match, which occurs if $S$ does not clash with $R$ but is not subsumed by it, *i.e.*, $\mathcal{T} \models R \sqcap S \not\sqsubseteq \bot$ and $\mathcal{T} \models S \not\sqsubseteq R$. The output of CA consists of a concept $H \in \mathcal{ALN}$ such that $\mathcal{T} \models S \sqcap H \sqsubseteq R$ and $S \sqcap H$ is satisfiable in $\mathcal{T}$. It should be noted CA and the other inference services for semantic matchmaking in Tiny-ME are defined under the Open World Assumption [30], *i.e.*, missing information in a concept expression is not equivalent to negation, but it simply represents an unspecified constraint, *e.g.*, because unknown or deemed irrelevant. The solution $H$ (for *Hypothesis*) can be interpreted as what is requested in $R$ and not specified in $S$, providing an explanation for missed Subsumption and a way to move from a potential to a *full match* (a.k.a. *subsume match* [28,29]), which is the desired outcome of the matchmaking framework and occurs when $S \sqsubseteq R$, *i.e.*, all features in the request are provided by the resource[6]. Having $R$ and $S$ unfolded and CNF-normalized concept expressions in $\mathcal{ALN}$, the structural algorithm for $CA(R, S)$ is:

1. set $H := \top$;
2. for each (possibly negated) atomic concept $A$ in $R$, if $\nexists B$ in $S$ s.t. $B \sqsubseteq A$, then put $A$ in conjunction with the concept expression for $H$;
3. for each role $P$ such that $\geq xP$ is in $R$, if $\geq yP$ is not in $S$ or $\geq yP$ is in $S$ with $y < x$, then put $\geq xP$ in conjunction with the concept expression for $H$;
4. for each role $P$ such that $\leq xP$ is in $R$, if $\leq yP$ is not in $S$ or $\leq yP$ is in $S$ with $y > x$, then put $\leq xP$ in conjunction with the concept expression for $H$;
5. for each role $P$ s.t. $\forall P.E$ is in $R$ and $\forall P.F$ is in $S$, then compute Abduction recursively on the fillers: $H' = CA(E, F)$ and put $\forall P.H'$ in conjunction with the concept expression for $H$.

For example, referring to the above $TB_{ex}$, $H = CA(A, B) = \geq 3\ P$.

**Concept Bonus** (CB) is also useful in matchmaking settings, since a resource $S$ could contain features not requested in $R$ – possibly because the requester was not aware of them or did not care – which could be exploited in a query refinement process. CB extracts and returns a *Bonus* concept $B$ from $S$, denoting what the

[6] *Exact match*, occurring when $S \equiv R$, is obviously the best possible outcome, but full match is equally desirable from the point of view of requesters, since all their preferences are met.

resource provides even though the request did not ask for it. The algorithm for finding the Bonus $B$ of $S$ w.r.t. $R$ is the same for the CA problem where $R$ and $S$ are swapped, *i.e.*, $R$ acts as resource and $S$ as request.

The $\mathcal{ALN}$ CNF for concept expressions induces the definition of a metric space with a *norm* operator $\|\cdot\|$. In the matchmaking framework supported by Tiny-ME, the CNF norm of $G$ and $H$ then represents a semantic distance **penalty** for CC and CA, respectively, used to rank resources w.r.t. a given request. Similarly, $\|B\|$ provides a relevance measure for the Bonus. In [30] penalty values have been proposed where:

1. each (possibly negated) atomic concept counts as 1;
2. modified number restrictions are weighted as the ratio of the difference of cardinality in $R$ and $S$ on the same role w.r.t. the cardinality value in $R$ (if $R$ lacks a number restriction on that role, the penalty is 1);
3. universal restriction fillers computed recursively are counted like above.

For CA, CB, and CC, the summarized algorithms aim to a minimality criterion, since one usually wants to hypothesize or give up as little as possible. Conversely, a maximality criterion is adopted for the **Concept Difference** reasoning service (CD), which defines a way to subtract information in a concept description from another one: in the original definition by Teege [40], if $\mathcal{T} \models R \sqsubseteq S$, then the output of difference $D = CD(R, S)$ is a concept $D \in \mathcal{ALN}$ such that $R \equiv S \sqcap D$. In that case, one typically wants to subtract as much as possible. In Tiny-ME, the applicability of CD has been extended from the full match (*i.e.*, subsumption) case to potential and partial matches as well, by exploiting CC and CB. In detail, given two unfolded and CNF-normalized $\mathcal{ALN}$ concept expressions $R$ and $S$, $CD(R, S)$ is computed structurally as in what follows:

1. if $R \sqcap S \sqsubseteq \bot$, *i.e.*, $R$ and $S$ are not consistent, then use Concept Contraction to retrieve the part $K$ of $S$ that is consistent with $R$. Otherwise, $K := S$.
2. return the Concept Bonus between $K$ and $R$: $D := CB(R, K)$.

For example, referring again to $TB_{ex}$, $CD(B, A) = B$ and $CD(A, B) = \geq 3\ P$.

While CA, CB, CC, and CD are useful in one-to-one discovery, matchmaking and negotiation scenarios, Tiny-ME also includes the **Concept Covering** (CCov) non-standard inference for many-to-one composition of a set of elementary instances to answer complex requests [8]. Basically, CCov takes a set of resources and a request and aims to (i) cover (*i.e.*, satisfy) features expressed in the request as much as possible through the conjunction of resources, and (ii) provide an explanation of the possibly uncovered part. In formulae, given a concept expression $R$ (request) and a set of concept expressions $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ in a KB (available resources), where $R$ and $S_1, S_2, \ldots, S_n$ are satisfiable in the reference ontology $\mathcal{T}$, the output of CCov is a pair $\langle \mathcal{S}_c, H \rangle$ where $\mathcal{S}_c \subseteq \mathcal{S}$ contains concepts in $S$ forming a (possibly incomplete) covering of $R$ w.r.t. $\mathcal{T}$ and concept $H$ is the (possible) part of $R$ not covered by elements in $\mathcal{S}_c$. The structural algorithm for CCov, given $\{R, S_1, S_2, \ldots, S_n\}$ unfolded and CNF-normalized, is:

1. set $\mathcal{S} := \emptyset$ and $H := R$;
2. repeat the following steps until $S_{max} \equiv \top$:
   (a) set $r_{min} := \|H\|$ and $S_{max} := \top$;
   (b) for each $S_i$ in $\mathcal{S}$, if $S_i \sqcap R \not\sqsubseteq \bot$ (*i.e.*, $S_i$ and $R$ are Consistent) and $CD(S_i, H) \not\equiv \top$ (*i.e.*, $S_i$ covers $H$), then compute $H_i, r_i := CA(H, S_i)$. If $r_i < r_{min}$, update $r_{min} := r_i$, $S_{max} := S_i$ and $H_{max} := H_i$;
   (c) if $S_{max} \not\equiv \top$, add $S_{max}$ to $\mathcal{S}_c$, remove it from $\mathcal{S}$, and set $H := H_{max}$;
3. return $\mathcal{S}_c, H$.

In step 2, each iteration of the loop computes $H_i$ and $r_i$ respectively as the CA hypothesis and penalty with respect to the remaining uncovered part $H$. The resource with minimal penalty, *i.e.*, with maximal covering of $H$, is added to the set $\mathcal{S}_c$, until no resources further increase the covering.

Tiny-ME can be also exploited in more general knowledge-based applications, as it provides *Classification* and *Coherence* services over ontologies.

**Ontology Classification** computes the overall concept taxonomy induced by the subsumption relation, from $\top$ (*Top* a.k.a. *Thing*) to $\bot$. The system adopts a variant of the *enhanced traversal* algorithm in [41], with a number of optimizations, such as caching of subsumption check results, exploitation of told subsumers and disjoints [39], and caching of unfolded and normalized concept expressions [11].

**Ontology Coherence** involves checking that all named concepts in the TBox are satisfiable [42]. As in [11], instead of computing it directly by checking the satisfiability of all TBox concepts, Tiny-ME adopts a variant of the Classification algorithm that stops as soon as an unsatisfiable concept is detected. This is preferred over the naive approach, as most of the runtime of structural inference algorithms is spent in unfolding and CNF normalization. Since Classification avoids Subsumption checks (and thus unfolding and normalization) as much as possible, this generally results in significantly improved performance.

### 3.2. Architecture

**High-level architecture.** The overall high-level architecture is reported in Fig. 2 and described in what follows:

- *Core layer:* it is written in standard C11 with no compiler extensions nor platform-specific API calls, and contains a highly optimized implementation of standard and non-standard inference algorithms, along with their backing data structures. This layer has been carefully designed to be independent from the way knowledge is represented or stored: $\mathcal{ALN}$ OWL entities (*i.e.*, named constructs such as classes, object properties, and named individuals) are encoded as numerical identifiers, called *entity pointers*, and their string representation is never required during reasoning. This feature is exploited by the *Axiom provider* interface the reasoner queries to retrieve structured representations of KB axioms. The Core layer provides an optional *logging* API, querying the *String provider* interface for the string representations of entity pointers. In a distributed SWoE architecture, some computing devices may not need string representation capabilities, or may not have enough memory to deal with it: in that case, they can just implement the Axiom provider API and still be able to carry out reasoning tasks on (unlabeled) entities.
- *Platform-specific APIs:* the modularity of the architecture allows for multiple APIs implemented in different programming languages, as described in Section 3.4. In general, the adoption of C11 for the reasoning core enables a wide variety of potential higher-level APIs, as the runtimes of most programming languages offer at least some basic interoperability with C code.

**Core architecture.** The reasoning core can be compiled both as static or dynamic linking library, and can run on any platform for which a C compiler exists. It is important to note that, while C lacks object-orientation, it is still possible to build components that have high cohesion by logically bundling structured data and functions that operate on them. This approach has been

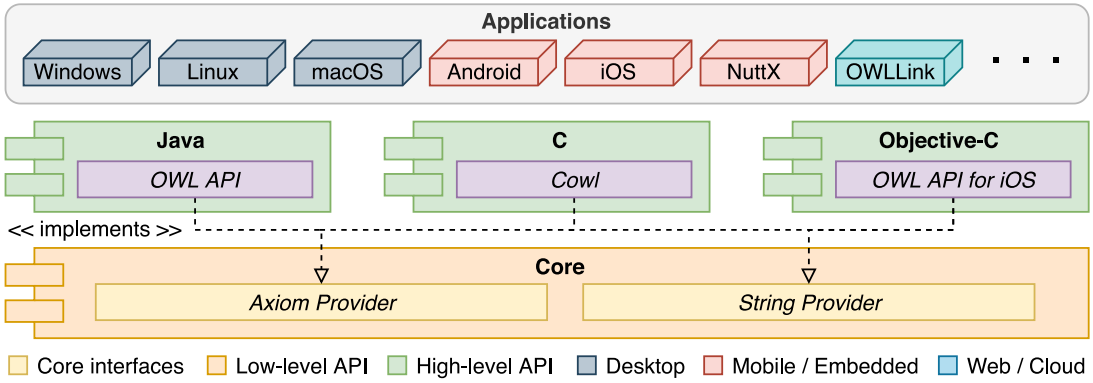


**Fig. 2.** High-level architecture.

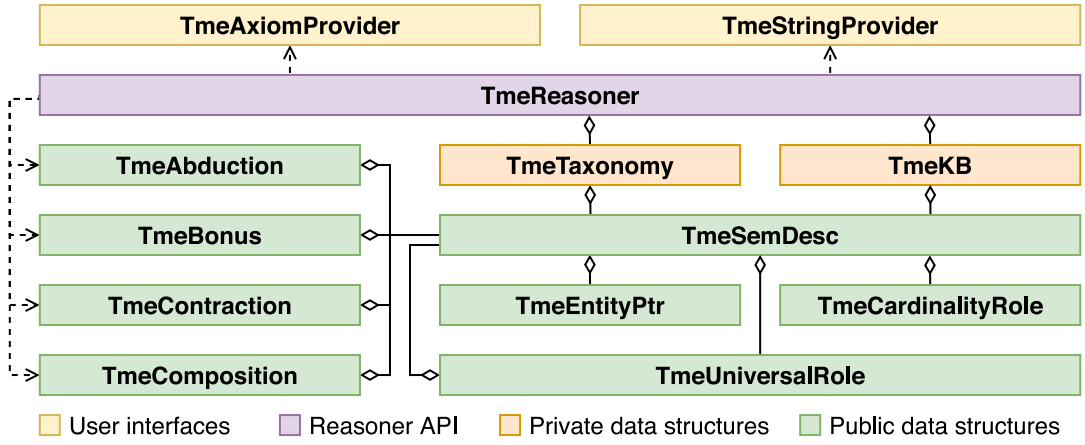


**Fig. 3.** Core architecture.

adopted throughout the core, resulting in a highly modular and easily maintainable codebase. The main components of the core architecture are illustrated in Fig. 3 and described hereafter:

- `TmeAxiomProvider`: retrieval of axioms from KBs is abstracted away by means of this interface. Implementors must essentially map $\mathcal{ALN}$ OWL class expressions to `TmeSemDesc` structures, described hereafter.
- `TmeStringProvider`: returns string representations of entity pointers, making it possible to visualize class expressions when using the built-in logging API.
- `TmeReasoner`: implements reasoning tasks over ontologies, namely Classification and Coherence check, and supplies a facade API to inference services on class expressions, which are in turn provided by lower-level components.
- `TmeKB`: exposes KB management primitives, which mostly involve loading and preprocessing class expressions via unfolding and CNF normalization. Both are *lazy*: an internal cache keeps track of whether a concept has been only unfolded, or both unfolded and normalized, in order to avoid unnecessary computations [11].
- `TmeTaxonomy`: allows manipulating the concept hierarchy resulting from Classification, supporting insertion, deletion, merging, and retrieval of ancestors and successors of classes.
- `TmeSemDesc`: the numerical representation of an $\mathcal{ALN}$ class expression in CNF. It models the conjunction of $C_{CN}$, $C_{\geq}$, $C_{\leq}$, $C_{\forall}$ components storing (possibly negated) atomic classes and minimum cardinality, maximum cardinality, and universal object property restrictions, respectively. Class expression elements are stored in vectors, whose type depends on the kind of atom. In particular, atomic classes and their negation are represented by `TmeEntityPtr`, a `typedef` for an integer type; `TmeCardinalityRole` models unqualified number restrictions with a property identifier (`TmeEntityPtr`) and a cardinality (of integer type); universal quantifiers are represented by `TmeUniversalRole`, using an integer type for the property identifier and a pointer to the filler. The whole class description is therefore made of just integers, allowing for a compact memory representation and lower computational overhead.
- `TmeAbduction`, `TmeContraction`, `TmeBonus`, `TmeComposition`: model the result of CA, CB, CC, and CCov, respectively. All these structures also include a penalty score, as explained in Section 3.1.

### 3.3. Non-standard inference services and OWLlink extension

In order to support client–server interaction in Web contexts as well as in microservice architectures for Cloud and Edge Computing, *Tiny-ME* leverages and extends the standard OWLlink protocol [9]. OWLlink provides a declarative OWL reasoner interface for asserting axioms in KBs and requesting standard inferences. Tiny-ME supports the Subsumption, Satisfiability, Classification, and Consistency/Coherence check. Furthermore, a novel OWLlink extension[7] is proposed here (following the official protocol extension guidelines [43]) to support non-standard reasoning. Based on the inference definitions recalled in Section 3.1, new request

[7] XML Schema available at http://swot.sisinflab.poliba.it/tinyme/#download.

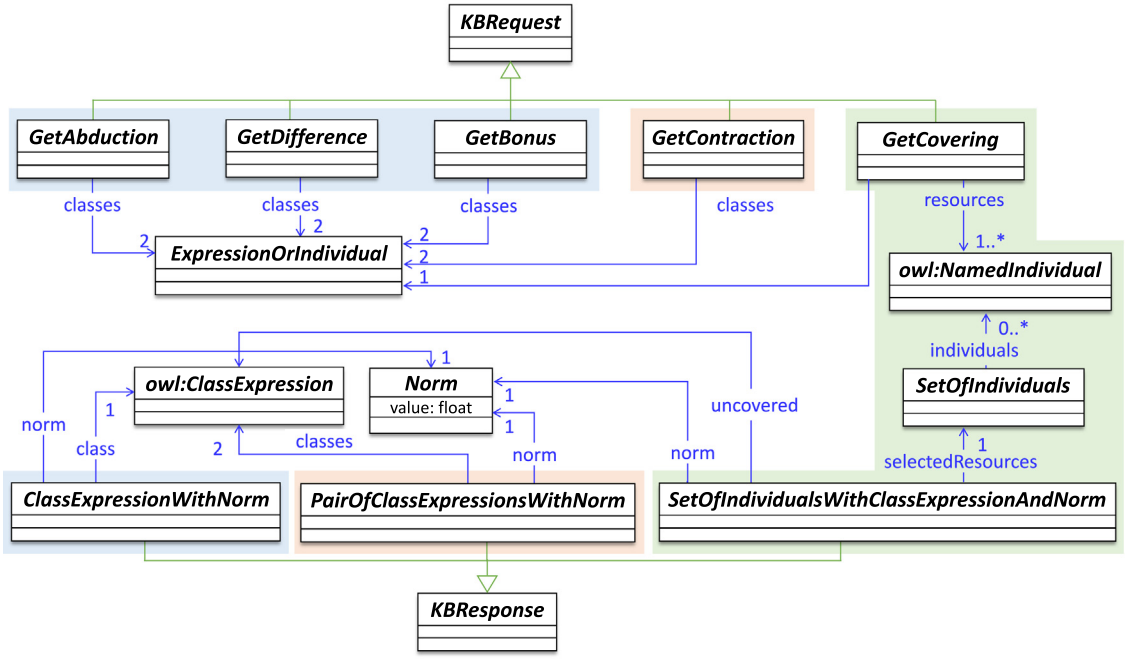


**Fig. 4.** OWLlink extension – Request and response objects.

types and their related replies (highlighted with the same color in the diagram in Fig. 4) have been defined, along with the HTTP/XML binding:

- *GetAbduction, GetBonus, GetDifference*: used to invoke CA, CB, and CD, respectively. These messages require two *ExpressionOrIndividual* arguments, corresponding to either an OWL class expression or a named individual in the reference KB, *i.e.*, a *request* $R$ and a *resource* $S$. Replies to these inferences consist of: (i) an OWL class expression, representing the uncovered part of a request in CA, the additional information provided by a resource in CB or the remaining expression after a CD, respectively; (ii) a non-negative *penalty score*, that is the semantic distance of $R$ from $S$, computed as the CNF norm of the returned class expression, to be intended as the *explanation* of the numerical result.
- *GetContraction*: also in this case, the message requires two *ExpressionOrIndividual* arguments modeling a request $R$ and a resource $S$. The response object consists of two OWL class expressions, *i.e.*, the conflicting requirements $G$ (Give up) and the contracted (compatible) version $K$ (Keep) of $R$, with a penalty score measuring the incompatibility degree of between $R$ and $S$.
- *GetCovering*: the request for a CCov non-standard reasoning service includes an *ExpressionOrIndividual* argument $R$ and one or more OWL individuals in the KB acting as available resources. This service replies with the subset of the input resources able to cover the request as much as possible, together with an OWL class expression of the uncovered part (possibly $\top$) and a penalty score evaluating the part of $R$ that has not been covered.

A fork of the Java-based *OWLlink API*[8] [44] compatible with OWL API 5 has been developed to implement the extended interface[9]. It should be pointed out that Tiny-ME currently does not support `Tell` OWLlink requests to assert axioms to a KB; they are left for a future update. `LoadOntologies` requests are supported, instead, for loading a KB from a URL. A *Dockerfile* is available on Tiny-ME's homepage to build a Docker container featuring the reasoner working as an OWLlink server.

### 3.4. Platform-specific APIs

The proposed reasoner currently implements the following platform-specific APIs:

- **C:** the native interface of the system. It exposes the public API of the reasoning core, while implementing the axiom and string providers by means of the *Cowl*[10] library. Cowl is a lightweight C11 interface for a complete OWL 2 data model and provides support for deserializing ontologies in functional-style syntax [45]. This solution can be used on any platform providing a C compiler and should be preferred for performance-critical and embedded software development, for which C is currently the most adopted language [46].
- **Java:** the interface for Java SE and Android runtime environments. Axiom and string providers are implemented through the OWL API [6]. The data model is basically a mapping of Java classes and methods to the above C structures and related functions through the *Java Native Interface* (JNI).[11] As an example, inference services are provided via the `Reasoner` class, which wraps the native `TmeReasoner` structure and implements OWL API's `OWLReasoner` interface; class expressions are modeled by the `SemanticDescription` class, which maps the native `TmeSemDesc` structure, and a similar approach is adopted for other logic constructors. Special care has been devoted to the management of native memory: Java objects backed by native

[8] OWLlink OWL API adaptor: https://github.com/ignazio1977/owllink-owlapi.
[9] OWLlink matchmaking extension Git repository: https://github.com/sisinflab-swot/owllink-matchmaking-extension.
[10] Cowl home: http://swot.sisinflab.poliba.it/cowl/.
[11] Java Native Interface documentation: https://docs.oracle.com/en/java/javase/13/docs/specs/jni/index.html.

structures are tracked by the `NativeMemoryManager`, exploiting *phantom references*[12] to trace when they are about to be garbage-collected so as to invoke native deallocators.

- **Objective-C:** the preferred interface for iOS and macOS applications. The axiom and string providers are implemented using the OWL API for iOS [7]. Objective-C class instances and methods map lower-level C structures and functions, although the wrapping logic is thinner than that of the Java API: since Objective-C is a strict superset of C, no additional interfaces are required, and this results in simpler architecture and improved performance (see Section 5). Memory management is also greatly simplified, as the presence of *Automatic Reference Counting* (ARC)[13] and reliable finalizers[14] allows tying the lifetime of C allocations to the Objective-C wrappers.

*Tiny-ME* also runs on Linux-based *Docker* containers with either the C or the Java API. No modification is needed, and the reasoner can be invoked remotely through OWLlink.

## 4. Case study: hazard risk detection on Pixhawk embedded drone autopilots

In order to validate the portability of the proposed reasoner and its suitability to SWoE resource-constrained devices, an experimental installation has been attempted to the *3DR IRIS+*[15] UAV (a.k.a. drone), equipped with a *Pixhawk 1*[16] autopilot (*i.e.*, flight controller). This unit is equipped with a *STM32F427*[17] system-on-chip, including a 180 MHz ARM Cortex M4 CPU and 256 kB of SRAM. The autopilot runs the *PX4*[18] FMUv2 firmware on the Apache NuttX RTOS, supporting the development of user-defined applications and modules.[19]

Basically, development requires configuring the *CMake* build system on the computer used for cross-compilation, by specifying the location of source files and instructing it to build and deploy the modified firmware to the Pixhawk, to be connected via USB, as shown in Fig. 5.

The C interface of Tiny-ME has been embedded in a command-line application runnable through the NuttX shell. The port has just required configuring the build system: notably, no source code changes have been needed to successfully compile the reasoner. Runtime tests have shown the correctness of the operation.

Subsequently, in order to assess on-board reasoning feasibility and usefulness of available inferences and language expressiveness in a representative scenario, a case study has been conducted on UAV-based detection of fire and explosion risk from gas or vapor. According to the European Union (EU) Directive 2014/34/UE,[20] risk exists if the following conditions are true: (i) concentration is higher than the substance-specific *Lower Explosion Limit* (LEL), defined as the lowest value able to produce fire in the presence of an ignition source; (ii) for a gas, oxygen concentration is higher than the *Limiting Oxygen Concentration* (LOC), defined as the value below which combustion cannot occur; (iii) for a vapor, air temperature is higher than the substance-specific *flashpoint* threshold.

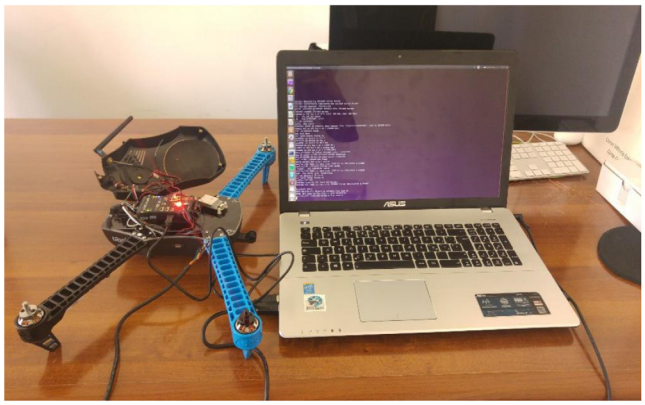

**Fig. 5.** Testbed setup.

Fig. 6 shows the upper-level classes of the OWL $\mathcal{ALN}$ KB[21] (292 axioms, 73 classes, 13 object properties and 27 individuals) modeled for the case study. The UAV is endowed with a global navigation satellite system antenna as well as sensors for temperature, atmospheric pressure, wind speed, oxygen concentration, and the concentration of each substance to be monitored. A periodic task has been scheduled on the UAV to: (i) collect values from on-board sensors and preprocess them; (ii) exploit Tiny-ME to compose a class expression describing relevant quantities for each monitored substance; (iii) perform semantic matchmaking w.r.t. KB individuals representing risk conditions. Basically, the main thread gathers values from sensors and preprocesses them, based on a set of thresholds and other simple filters. For each piece of data, Tiny-ME is used to instantiate a conjunct referring to the class of the ontology corresponding to the data value and then it is exploited again to compose a conjunctive expression. Finally, reasoning is invoked for semantic matchmaking. In Fig. 6, classes of explosive or flammable atmosphere conditions for the considered substances are highlighted in blue, while environmental features which influence risk levels are highlighted in green. For example, let us consider a 6 g/m$^3$ methane concentration, 1 m/s wind speed, and 15% oxygen concentration: based on class definitions in the ontology as derived from the EU directive, this produces the following annotation (reported in Manchester syntax [47]):

*R*: `MediumConcentration_Methane and HighOxygenConcentration_Methane and LowVentilation_Methane`.

As said before, by relying on a set of predefined thresholds, Tiny-ME instantiates a conjunct for every monitored substance, each referring to one environmental feature class in the ontology. Hence, classes appearing in *R* are defined in the ontology as:

```
MediumConcentration_Methane ≡ Methane and (hasConcentration only g/m3) and (hasConcentration min 6 owl:Thing) and (hasConcentration max 11 owl:Thing).
HighOxigenConcentration_Methane ≡ Methane and (withOxygenConcentration some owl:Thing) and (withOxygenConcentration only ((hasOxygenConcentration only percent) and (hasOxygenConcentration min 14 owl:Thing))).
```

[12] PhantomReference documentation: https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/ref/PhantomReference.html.

[13] ARC documentation: https://clang.llvm.org/docs/AutomaticReferenceCounting.html.

[14] NSObject documentation: https://developer.apple.com/documentation/objectivec/nsobject.

[15] IRIS+ home: https://www.arducopter.co.uk/iris-quadcopter-uav.html.

[16] Pixhawk 1 home: https://docs.px4.io/v1.9.0/en/flight_controller/pixhawk.html.

[17] STM32F427 home: https://www.st.com/en/microcontrollers-microprocessors/stm32f427-437.html.

[18] PX4 home: https://px4.io.

[19] PX4 developer documentation: https://dev.px4.io.

[20] Directive 2014/34/UE: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex:32014L0034.

[21] KB available at http://swot.sisinflab.poliba.it/onto/drone_risk_detection.owl.

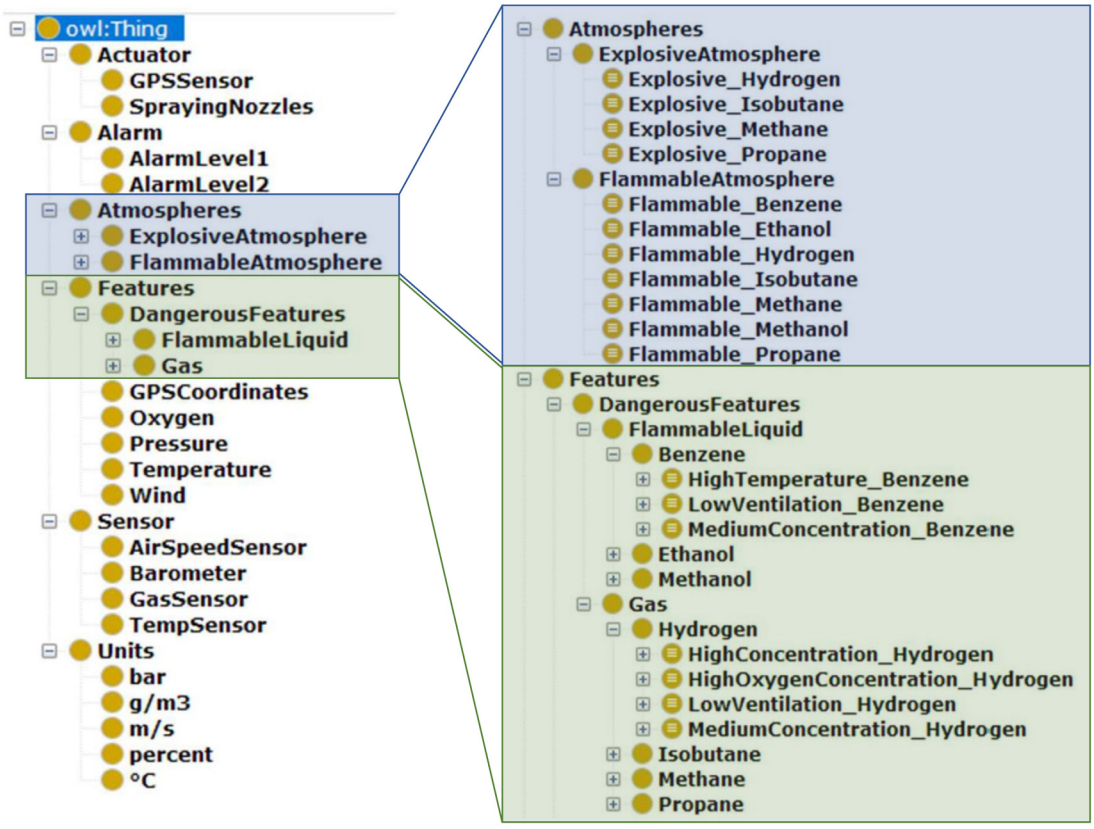


**Fig. 6.** Classes in the KB modeled for the case study.

`LowVentilation_Methane` ≡ `Methane and (withWindSpeed some owl:Thing) and (withWindSpeed only ((hasSpeed only m/s) and (hasSpeed max1 owl:Thing)))`.

As one may notice, Tiny-ME does not support datatype properties, since it is based on the $\mathcal{ALN}$ DL. Hence, the semantics of numerical data properties is approximated through unqualified number restrictions, treated in reasoning algorithms as described in Section 3.1. After the above semantic labeling, the UAV performs matchmaking of $R$ with all KB individuals that are instances of *Explosive_Atmosphere* and *Flammable_Atmosphere* subclasses, such as *Explosive_methane* and *Flammable_methane*. For each substance of interest, this stage allows inferring if conditions for fire or explosion are met, according to environmental parameters monitored and modeled in $R$. For each individual, first of all the UAV uses Tiny-ME to check if it is consistent with $R$, *i.e.*, if their conjunction is satisfiable. If the check fails, then surely the current situation does not pose a risk. Otherwise, Concept Abduction is invoked to compute the semantic distance $d$ between the monitored situation and the risk model represented by the specific individual. It is useful to recall that each KB individual is considered as a request in the matchmaking framework described in Section 3.1, while $R$ is treated as a resource. This entails that the semantic distance $d$ represents "how much" is missing from $R$ to fully satisfy risk conditions. The threshold $d = 4.0$ has been determined for the case study, based on the algorithm for penalty computation in Section 3.1 and the adopted KB modeling patterns. Below this threshold, a risk is recognized and the UAV issues an alert.

In particular, explosion risk has been tested for all substances before fire risk, as the former requires raising a higher-severity alert. Following up the above example, the KB contains these two risk profiles for methane:

1. *Explosive_methane*: `HighConcentration_Methane and HighOxygenConcentration_Methane and LowVentilation_Methane`
2. *Flammable_methane*: `LowVentilation_Methane and MediumConcentration_Methane and HighOxygenConcentration_Methane`

The compatibility check between $R$ and *Explosive_methane* fails, as class `HighConcentration_Methane` is defined as having a LEL of 12 g/m$^3$, while `MediumConcentration_Methane` has a value between 6 g/m$^3$ and 11 g/m$^3$: `HighConcentration_Methane` ≡ `Methane and (hasConcentration only g/m3) and (hasConcentration min 12 owl:Thing)`

The descriptions of the two classes have number restrictions with disjoint intervals on the same property, therefore they are disjoint; this implies that explosion risk is lacking. Conversely, *Flammable_methane* $\sqsubseteq R$ and CA detects a full match ($d = 0$), *i.e.*, semantic distance below the given threshold. As a consequence, the UAV will raise a fire alert related to the methane substance.

Loading the case study KB has required coping with strict Pixhawk 1 memory limitations, by means of two actions: (i) disabling all not-relevant modules from the default FMUv2 firmware configuration, *e.g.*, debugging and audio control; (ii) encoding the KB to *Protocol buffers* binary format[22] with the *nanopb* library,[23] which features a compact code size. This has not only allowed downsizing the KB from ∼ 85 kB (in OWL 2 functional syntax) to ∼ 2.1 kB, but also unloading the Cowl parser from memory. Early quantitative tests have been carried out; each test has been repeated 5 times and average results have been taken. Times have been measured through log messages in the code, memory

[22] Protocol buffers home: https://developers.google.com/protocol-buffers.
[23] Nanopb repository: https://github.com/nanopb/nanopb.

occupancy via the `free` command in the NuttX shell. Results are as follows: KB loading has taken $117.2 \pm 7.22$ ms; the annotation and matchmaking task on 1 substance has taken $10.2 \pm 1.12$ ms, whereas on all the 8 monitored substances in the KB $55.6 \pm 2.58$ ms; used memory after reasoner and KB loading, before starting the main detection loop, has been $151.5 \pm 0.21$ kB; used memory at the end of one detection loop iteration has been $176.0 \pm 0.25$ kB. Basically, these results suggest the computational sustainability of the proposed tool in a typical ubiquitous computing scenario on a very resource-constrained device.

## 5. Performance evaluation

An early experimental campaign has been carried out to evaluate the computational performance of Tiny-ME in Ontology Classification and non-standard inference tasks. Tests have been executed on desktop and mobile platforms by means of the EVOWLUATOR [48] framework.[24] The desktop testbed is an *Apple Mac Mini (2014)*[25], while mobile tests have been carried out on an *Apple iPhone 7*[26], and a *HTC/Google Nexus 9* tablet.[27]

In what follows, *peak memory usage* refers to the *maximum resident set size* (MRSS) of the reasoner process, measured by means of the `getrusage` POSIX call on iOS and Android, and via the cross-platform `psutil`[28] Python library on desktop platforms.

All performance results are the average of five cold runs. Standard deviations for time and memory results are not reported to avoid clutter in tables and plots, as they are consistently small (about 5% and 1% of the mean values, respectively).

Full results are available at a permanent URL[29] and can be reproduced using the setup instructions available on the project Web page. Results also include correctness evaluation, which has been carried out using Konclude and Mini-ME Swift as test oracles respectively for Ontology Classification and the non-standard Matchmaking task described in Section 5.2. All Tiny-ME variants have provided correct and complete inferences for all ontologies and reasoning tasks.

### 5.1. Classification

The dataset exploited for the classification test consists of 1364 knowledge bases obtained from the 2014 *OWL Reasoner Evaluation Workshop* competition[30] out of the 16555 KBs in the DL classification corpus (8.24%), considering all the KBs supported by Tiny-ME (*i.e.*, having at most $\mathcal{ALN}$ as indicated expressiveness, without general concept inclusions and other unsupported logic constructors). As said, tests refer to both desktop and mobile platforms.

**Desktop.** Konclude has been considered as reference reasoner, due to previous campaigns [11,49] indicating it is the most reliable and high-performance reasoner with respect to Ontology Classification. Basically, it has been selected as an oracle for inference correctness, whilst the performance report does not imply a direct comparison with Tiny-ME, as the two systems are grounded on DLs with different expressiveness and diverse feature sets. However, since no other actively developed reasoner targets the $\mathcal{ALN}$ DL specifically (and a fair comparison is only possible with Mini-ME), Konclude performance has to be taken into consideration when assessing the basic goal of this work, *i.e.*, that the architecture and optimizations of Tiny-ME make it a worthwhile option for SWoE resource-constrained devices. Figs. 7(a) and 7(b) depict the overall time (cumulative parsing and reasoning time) and memory peak as a function of ontology size. The comparison involves Tiny-ME with C, Java SE and Objective-C (ObjC) interfaces, Mini-ME 2.0 for Java SE and Mini-ME Swift for macOS.

Aggregated performance metrics, such as those reported in Table 3, refer to the set of 1168 ontologies that all reasoners were able to classify correctly within a 20 min timeout (wall-clock time) and with no runtime errors. This is required to allow for a fair comparison of cumulative execution times. Conversely, scatterplots show all data points for all reasoners.

As highlighted in Table 3, Tiny-ME C has outperformed all the other systems with respect to both time and memory usage. The gap is particularly evident for ontologies smaller than 1 MB, which is a relevant outcome for resource-constrained SWoE scenarios.

The Objective-C API behaves better than the Java one, as expected due to thinner Objective-C wrappers than those necessary to interface with the JNI, as explained in Section 3.4. Tiny-ME ObjC and Mini-ME Swift are fairly close, with the former having slightly better performance: even though they share the same OWL parser and data model, Tiny-ME ObjC benefits from the more efficient C core. Mini-ME 2.0 and Tiny-ME Java SE exhibit a similar trend in terms of processing time and memory occupancy for small ontologies, but they diverge when the input size grows (see Table 3). Both systems use the OWL API 3.x library for ontology parsing, which contributes the most to the overall classification time for smaller ontologies; as the size grows, reasoning time becomes dominant and the benefits of the C core get evident. The analysis of memory peaks produces analogous findings, reported in Table 3.

**Mobile.** Fig. 8 plots classification results obtained by Mini-ME and Tiny-ME variants built and run on the Android and iOS operating systems. For both mobile platforms, performance analysis confirms observations made for the desktop tests. Reasoners running on iOS were able to process all the ontologies in the dataset within the imposed timeout and without errors, while Android reasoners started running out of memory while classifying ontologies larger than about 20 MB, as highlighted by both Fig. 8 (missing data points can be noticed for the largest ontologies on Android reasoners) and Table 3 (smaller maximum memory peaks for both Android reasoners than both iOS ones, due to out-of-memory errors).

### 5.2. Semantic matchmaking

All Tiny-ME variants (desktop and mobile) have been also evaluated about non-standard inference tasks, using four $\mathcal{ALN}$ KBs summarized in Table 4. Following the approach in [11], a matchmaking task starts with a *compatibility* check between a $\langle request, resource \rangle$ pair: CA is performed in case their conjunction is satisfiable, otherwise CC is executed, followed by CA on the contracted version of the request.

Fig. 9 recalls main outcomes. It should be noted that ontology size varies among reasoners using the OWL API for iOS and Cowl, which only support the RDF/XML and functional serializations, respectively. The latter has been also adopted for reasoners using the Java OWL API. Results are similar to those of Classification tests: Tiny-ME C significantly outperforms the other reasoners; Mini-ME Swift and Tiny-ME Obj-C have similar behaviors, and they both operate better than Java-based implementations. Fig. 9(b) highlights lower turnaround times on

[24] EVOWLUATOR Git repository: http://swot.sisinflab.poliba.it/evowluator/.

[25] Intel i7 4578u dual-core CPU at 3.0 GHz, 16 GB DDR3 RAM at 1600 MT/s, 1 TB HDD + 128 GB SSD (Fusion Drive), macOS Mojave 10.14.5.

[26] 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.

[27] Nvidia Tegra K1 dual-core CPU at 2.3 GHz, 2 GB LPDDR3 RAM at 1600 MT/s, 32 GB flash storage, Android 7.1.1 Nougat, patch level 5 October 2017.

[28] Python system and process utilities: https://psutil.readthedocs.io.

[29] https://zenodo.org/record/4405253.

[30] http://dl.kr.org/ore2014.

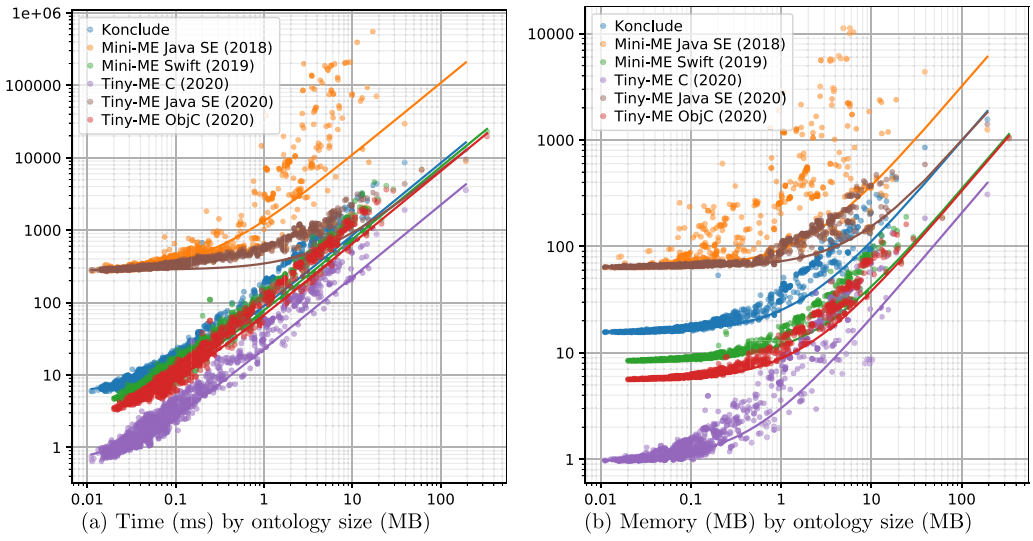


**Fig. 7.** Classification processing time and memory peak results on desktop.

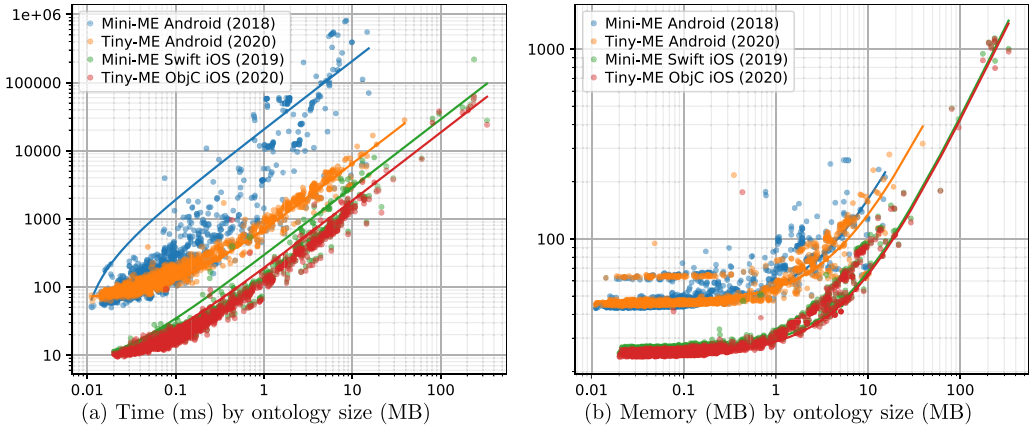


**Fig. 8.** Classification processing time and memory peak results on mobile.

**Table 3**
Dataset-wide evaluation results for classification.

| | | Incorrect results | Runtime Errors | Time-outs | Parsing Time (s) | Classif. Time (s) | Min Mem. Peak (MB) | Max Mem. Peak (MB) |
|---|---|---|---|---|---|---|---|---|
| Reasoners running on desktop | Konclude | n/a | 0 | 0 | 82.16 | 64.37 | 15.53 | 1559.52 |
| | Mini-ME Java SE | 136 | 0 | 60 | 443.01 | 4516.83 | 62.49 | 11237.72 |
| | Mini-ME Swift | 0 | 0 | 0 | 131.80 | 30.08 | 8.41 | 1065.92 |
| | Tiny-ME C | 0 | 0 | 0 | 27.26 | 8.41 | 0.96 | 307.55 |
| | Tiny-ME Java SE | 0 | 0 | 0 | 447.96 | 67.02 | 62.98 | 1408.6 |
| | Tiny-ME ObjC | 0 | 0 | 0 | 131.14 | 12.31 | 5.62 | 1032.87 |
| Reasoners on mobile platforms | Mini-ME Android | 132 | 100 | 5 | 326.71 | 8084.61 | 43.09 | 259.88 |
| | Tiny-ME Android | 0 | 8 | 1 | 343.05 | 96.54 | 44.18 | 133.71 |
| | Mini-ME Swift iOS | 0 | 0 | 0 | 450.07 | 344.95 | 24.94 | 1140.18 |
| | Tiny-ME ObjC iOS | 0 | 0 | 0 | 446.41 | 113.68 | 23.95 | 1117.37 |

**Table 4**
Features of KBs used for non-standard tests.

| | | Toy | Agriculture | Building | MatchAndDate |
|---|---|---|---|---|---|
| Knowledge Base 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 |

Android than on Java SE, because of the overall time including parsing, which is slower on the Mac Mini compared to the Nexus 9 (the latter has faster mass storage). Fig. 9(c) shows that the Android and Java SE variants have similar memory requirements, with slightly higher peaks reached on Java SE: this is presumably because the garbage collector is triggered more frequently

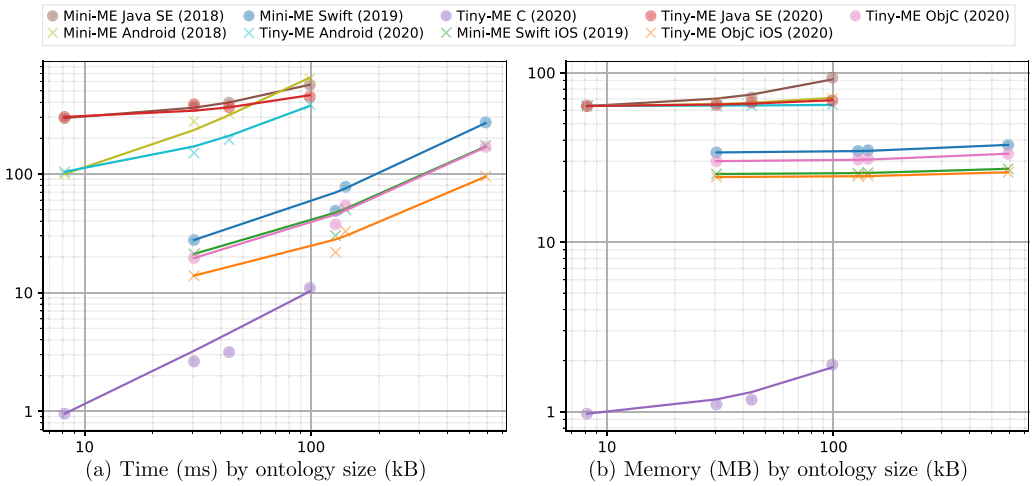


**Fig. 9.** Overall processing time and memory peak for the matchmaking task.

on mobile, as a consequence of stricter resource management policies. The opposite trend is observed on iOS because, as stated in [11], iOS reasoner variants are affected by a systematic memory overhead due to their graphical user interface, while desktop ones can run as command-line tools.

## 6. Conclusion and future work

The paper has presented Tiny-ME, a multiplatform lightweight OWL reasoner and matchmaker for the Semantic Web of Everything. It is freely available for academic evaluation. Tiny-ME supports standard and non-standard inference tasks in an OWL 2 subset corresponding to the $\mathcal{ALN}$ DL. Compared with the state of the art, it benefits from: (i) portability across desktop, mobile and embedded OSs as well as Docker containers, achieved by means of a common C core and platform-specific APIs; (ii) ease of use via streamlined APIs; (iii) high efficiency due to design and implementation choices, making the component really suitable for very resource-constrained devices. As an illustrative example of that, the reasoner has been successfully run without source code modification on a Pixhawk flight controller with the Nuttx RTOS. Performance evaluation evidences improved reasoning time and memory usage w.r.t. Mini-ME variants. Furthermore, a comparison with Konclude has been carried out despite the differences in language expressiveness and features. This has not only allowed the verification of inference correctness, but also the validation of the fundamental claim of the paper: Tiny-ME's architecture and optimization make it a viable and worthwhile option for SWoE resource-constrained devices.

The proposed architecture has laid the groundwork for overcoming current limitations, which mainly concern language expressiveness. Datatype support as well as reasoning in OWL 2 $\mathcal{EL}$, $\mathcal{EL}^+$, and $\mathcal{EL}^{++}$ are the primary objectives, which require not only data structure extension but also new reasoning algorithms in the C core. By virtue of the novel architectural paradigm, however, the inference implementation effort will be needed just once, and all available APIs will benefit of novel reasoning capabilities in a straightforward way.

A further aspect of future investigation concerns the enhancement of KB compression techniques featuring the SWoE reasoning tools and frameworks. Our work with Protocol buffers in the case study and small-scale comparison of five encoding algorithms – including the EXI XML-oriented W3C Recommendation [50] and the RDF-specific HDT [51] – for storing OWL 2 annotations in Near-Field Communication (NFC) tags [52] highlight the need for deeper exploration.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgments

This work has been supported by Italian PON projects FURTHER (Future Revolutionary Technologies for More Electric Aircraft), Italy and NGS (New Satellites Generation Components), Italy. The authors are grateful to Arnaldo Tomasino for useful implementations.

## References

[1] Sophocles, Ajax, in: The Tragedies of Sophocles, Cambridge University Press, 1904, translated into English prose by Sir Richard C. Jebb.
[2] E. Dinc, M. Kuscu, B.A. Bilgin, O.B. Akan, Internet of everything: A unifying framework beyond Internet of Things, in: Harnessing the Internet of Everything (IoE) for Accelerated Innovation Opportunities, IGI Global, 2019, pp. 1–30.
[3] F. Scioscia, M. Ruta, Building a Semantic Web of Things: issues and perspectives in information compression, in: Proceedings of the 3rd IEEE International Conference on Semantic Computing, IEEE Computer Society, 2009, pp. 589–594.
[4] H.R. Arkian, A. Diyanat, A. Pourkhalili, MIST: Fog-based data analytics scheme with cost-efficient resource provisioning for IoT crowdsensing applications, J. Netw. Comput. Appl. 82 (2017) 152–165.
[5] B. Parsia, S. Rudolph, M. Krötzsch, P. Patel-Schneider, P. Hitzler, OWL 2 Web Ontology Language Primer, second ed., W3C Recommendation, W3C, 2012, http://www.w3.org/TR/owl2-primer.
[6] M. Horridge, S. Bechhofer, The OWL API: A Java API for OWL ontologies, Semant. Web 2 (1) (2011) 11–21.
[7] M. Ruta, F. Scioscia, E. Di Sciascio, I. Bilenchi, OWL API for iOS: early implementation and results, in: 13th OWL: Experiences and Directions Workshop and 5th OWL Reasoner Evaluation Workshop (OWLED - ORE 2016), in: Lecture Notes in Computer Science, vol. 10161, W3C, Springer, 2016, pp. 141–152.
[8] F. Scioscia, M. Ruta, G. Loseto, F. Gramegna, S. Ieva, A. Pinto, E. Di Sciascio, Mini-ME matchmaker and reasoner for the Semantic Web of Things, in: Innovations, Developments, and Applications of Semantic Web and Information Systems, IGI Global, 2018, pp. 262–294.
[9] T. Liebig, M. Luther, O. Noppens, M. Wessel, Owllink, Semant. Web 2 (1) (2011) 23–32.
[10] A. Steigmiller, T. Liebig, B. Glimm, Konclude: system description, J. Web Semant. 27 (2014) 78–85.
[11] M. Ruta, F. Scioscia, F. Gramegna, I. Bilenchi, E. Di Sciascio, Mini-ME matchmaker and reasoner for the semantic Web of Things, in: Innovations, Developments, and Applications of Semantic Web and Information Systems, IGI Global, 2018, pp. 262–294.

[12] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P. Patel-Schneider, The Description Logic Handbook, second ed., Cambridge University Press, 2002.
[13] F. Baader, S. Brandt, C. Lutz, Pushing the EL envelope, in: International Joint Conference on Artificial Intelligence, vol. 5, 2005, pp. 364–369.
[14] Y. Kazakov, M. Krötzsch, F. Simančík, The incredible ELK, J. Automat. Reason. 53 (1) (2014) 1–61.
[15] M. Ruta, T. Di Noia, E. Di Sciascio, G. Piscitelli, F. Scioscia, Semantic-based mobile registry for dynamic RFID-based logistics support, in: 10th International Conference on Electronic Commerce, ICEC 08, ACM Press, 2008, pp. 1–9.
[16] A. Sinner, T. Kleemann, Krhyper–in your pocket, in: International Conference on Automated Deduction, Springer, 2005, pp. 452–457.
[17] S. Ali, S. Kiefer, $\mu$OR–A micro OWL DL reasoner for ambient intelligent devices, in: 4th International Conference on Advances in Grid and Pervasive Computing, Springer, 2009, pp. 305–316.
[18] T. Kim, I. Park, S.J. Hyun, D. Lee, MiRE4OWL: Mobile rule engine for OWL, in: 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops, IEEE, 2010, pp. 317–322.
[19] S. Grimm, M. Watzke, T. Hubauer, F. Cescolini, Embedded $\mathcal{EL}$+ reasoning on programmable logic controllers, in: International Semantic Web Conference, Springer, 2012, pp. 66–81.
[20] W. Tai, J. Keeney, D. O'Sullivan, Resource-constrained reasoning using a reasoner composition approach, Semant. Web 6 (1) (2015) 35–59.
[21] C. Bobed, R. Yus, F. Bobillo, E. Mena, Semantic reasoning on mobile devices: Do Androids dream of efficient reasoners? J. Web Semant. 35 (2015) 167–183.
[22] Y. Kazakov, P. Klinov, Experimenting with ELK reasoner on android, in: 2nd International Workshop on OWL Reasoner Evaluation (ORE-2013), 2013, pp. 68–74.
[23] M. Ruta, F. Scioscia, G. Loseto, A. Pinto, E. Di Sciascio, Machine learning in the Internet of Things: A semantic-enhanced approach, Semant. Web 10 (1) (2019) 183–204.
[24] M. Ruta, F. Scioscia, E. Di Sciascio, Enabling the Semantic Web of Things: framework and architecture, in: Sixth IEEE International Conference on Semantic Computing (ICSC 2012), 2012, pp. 345–347.
[25] R. Yus, P. Pappachan, Are apps going semantic? A systematic review of semantic mobile applications, in: 1st International Workshop on Mobile Deployment of Semantic Technologies (MoDeST), 1506 of CEUR Workshop Proceedings, CEUR-WS, Aachen, Germany, 2015, pp. 2–13.
[26] D. Dell'Aglio, E. Della Valle, F. van Harmelen, A. Bernstein, Stream reasoning: A survey and outlook, Data Sci. 1 (1–2) (2017) 59–83.
[27] T. Di Noia, E. Di Sciascio, F. Donini, Semantic matchmaking as non-monotonic reasoning: A description logic approach, J. Artif. Intell. Res. (JAIR) 29 (2007) 269–307.
[28] L. Li, I. Horrocks, A software framework for matchmaking based on semantic web technology, Int. J. Electron. Commer. 8 (4) (2004).
[29] M. Paolucci, T. Kawamura, T.R. Payne, K. Sycara, Semantic matching of web services capabilities, in: International Semantic Web Conference, Springer, 2002, pp. 333–347.
[30] M. Ruta, E. Di Sciascio, F. Scioscia, Concept abduction and contraction in semantic-based P2P environments, Web Intell. Agent Syst. 9 (3) (2011) 179–207.
[31] C. Perera, A. Zaslavsky, C. Liu, M. Compton, P. Christen, D. Georgakopoulos, Sensor Search Techniques for Sensing as a Service Architecture for the Internet of Things, IEEE Sensors J. 14 (2) (2014) 406–420, http://dx.doi.org/10.1109/JSEN.2013.2282292.
[32] W. Van Woensel, S. Abidi, Optimizing semantic reasoning on memory-constrained platforms using the RETE algorithm, in: Extended Semantic Web Conference (ESWC), Springer, 2018, pp. 682–696.
[33] W. Van Woensel, N. Al Haider, A. Ahmad, S. Abidi, A cross-platform benchmark framework for mobile semantic web reasoning engines, in: International Semantic Web Conference, Springer, 2014, pp. 389–408.
[34] W. Van Woensel, S.S.R. Abidi, Benchmarking semantic reasoning on mobile platforms: Towards optimization using OWL2 RL, Semant. Web 10 (4) (2019) 637–663.
[35] J.-B. Lamy, Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies, Artif. Intell. Med. 80 (2017) 11–28.
[36] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, Z. Wang, HermiT: an OWL 2 reasoner, J. Automat. Reason. 53 (3) (2014) 245–269.
[37] J. Urbani, S. Kotoulas, J. Maassen, F. Van Harmelen, H. Bal, Webpie: A web-scale parallel inference engine using mapreduce, Web Semant.: Sci. Serv. Agents World Wide Web 10 (2012) 59–75.
[38] G. Schreiber, F. Gandon, RDF 1.1 XML syntax, W3C recommendation, W3C, 2014, http://www.w3.org/TR/rdf-syntax-grammar/.
[39] D. Tsarkov, I. Horrocks, FaCT++ description logic reasoner: System description, in: International Joint Conference on Automated Reasoning (IJCAR), Springer, Berlin, Germany, 2006, pp. 292–297.
[40] G. Teege, 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'94), ACM, 1994, pp. 540–550.
[41] F. Baader, B. Hollunder, B. Nebel, H.-J. Profitlich, E. Franconi, Am empirical analysis of optimization techniques for terminological representation systems, Appl. Intell. 4 (2) (1994) 109–132.
[42] M.O. Moguillansky, R. Wassermann, M.A. Falappa, An argumentation machinery to reason over inconsistent ontologies, in: G.R.S. Angel Kuri-Morales (Ed.), Advances in Artificial Intelligence–IBERAMIA 2010, Springer, Berlin, Germany, 2010, pp. 100–109.
[43] T. Liebig, M. Luther, O. Noppens, OWLlink: Structural Specification, W3C member submission, W3C, 2010, URL https://www.w3.org/Submission/2010/SUBM-owllink-structural-specification-20100701/.
[44] O. Noppens, M. Luther, T. Liebig, The OWLlink API: Teaching OWL components a common protocol, in: Proceedings of the 7th International Workshop on OWL: Experiences and Directions (OWLED 2010), 2010, pp. 13.1–13.4.
[45] B. Parsia, B. Motik, P. Patel-Schneider, OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax, second ed., W3C recommendation, W3C, 2012, http://www.w3.org/TR/owl2-syntax/.
[46] O. Hahm, E. Baccelli, H. Petersen, N. Tsiftes, Operating systems for low-end devices in the Internet of Things: a survey, IEEE Internet Things J. 3 (5) (2015) 720–734.
[47] M. Horridge, P. Patel-Schneider, OWL 2 Web Ontology Language Manchester Syntax, second ed., W3C note, W3C, 2012, http://www.w3.org/TR/owl2-manchester-syntax/.
[48] F. Scioscia, I. Bilenchi, M. Ruta, F. Gramegna, D. Loconte, A multiplatform energy-aware OWL reasoner benchmarking framework, J. Web Semant. 72 (2022) 100694.
[49] B. Parsia, N. Matentzoglu, R.S. Gonçalves, B. Glimm, A. Steigmiller, The OWL reasoner evaluation (ORE) 2015 competition report, J. Automat. Reason. 59 (4) (2017) 455–482.
[50] J. Schneider, T. Kamiya, D. Peintner, R. Kyusakov, Efficient XML Interchange (EXI) Format 1.0, second ed., W3C Recommendation, W3C, 2014, https://www.w3.org/TR/exi/.
[51] J.D. Fernández, M.A. Martínez-Prieto, C. Gutiérrez, A. Polleres, M. Arias, Binary RDF representation for publication and exchange (HDT), Web Semant.: Sci. Serv. Agents World Wide Web 19 (2013) 22–41.
[52] G. Loseto, F. Scioscia, M. Ruta, F. Gramegna, S. Ieva, A. Pinto, C. Scioscia, Knowledge-based decision support in healthcare via near field communication, Sensors 20 (17) (2020) 4923.