Automated reasoning for the semantic web of everything

Politecnico di Bari

Department of Electrical and Information Engineering

Electrical and Information Engineering Ph.D. Program

SSD: ING-INF/05 – Information Processing Systems

Final dissertation

Automated Reasoning for the Semantic Web of Everything

by

Ivano Bilenchi

Ph.D. program
Coordinator:
Prof. Mario Carpentieri

Supervisors:
Prof. Michele Ruta
Prof. Floriano Scioscia

36th Course, 01/11/2020 – 31/01/2024

Politecnico di Bari

Department of Electrical and Information Engineering

Electrical and Information Engineering Ph.D. Program

SSD: ING-INF/05 – Information Processing Systems

Final dissertation

Automated Reasoning for the Semantic Web of Everything

by

Ivano Bilenchi

Ph.D. program
Coordinator:
Prof. Mario Carpentieri

Supervisors:
Prof. Michele Ruta
Prof. Floriano Scioscia

Referees:
Prof. Stefano Mariani
Prof. Hasan Ali Khattak

36th Course, 01/11/2020 – 31/01/2024

In the pursuit of knowledge, every day something is acquired.
In the pursuit of wisdom, every day something is dropped.

- Laozi, Tao Te Ching [64]

To Giusy and Yuri.

Abstract

The evolution from the Internet of Things (IoT) into the Internet of Everything (IoE), driven by the miniaturization of IoT technology and the ever increasing connection of individuals, processes, and data through mobile and ubiquitous networks, represents a transformative change in how we interact with the digital world. In the IoE, living entities, objects, locations, and processes are interconnected and continuously generate streams of data, fuelling sophisticated analytics to derive actionable insights. While this shift enhances operational efficiency and data-driven decision-making, it also introduces new challenges, particularly in the realms of network bandwidth, power consumption, data protection, and privacy. In the IoE, intelligent information management becomes crucial to enable versatile autonomous processing and advanced, interoperable services for large-scale machine-to-machine and human-machine interactions.

Central to addressing these challenges is the concept of edge computing, which involves moving computational processes closer to data sources. This shift is pivotal for handling the volume and velocity of data generated by the IoE, and for mitigating the latency and bandwidth issues inherent in centralized processing models. One of the first incarnations of an intelligent information processing framework leveraging the edge computing paradigm is the Semantic Web of Things (SWoT), where ontology-based annotations of devices, objects, and phenomena are locally processed by ubiquitous intelligent agents via automated reasoning, facilitating autonomous actions towards specific goals.

The progression of the SWoT towards a Semantic Web of Everything (SWoE) mandates further integration of semantic technologies in a variety of pervasive computing interactions among people, things, processes, and data. This vision demands robust knowledge representation languages and automated inference capabilities, even on devices with strict processing, memory, and energy constraints. On-device local inference procedures are

critical in the SWoE, given the high volatility and limited reachability of more powerful devices.

Implementing the SWoE architecture poses significant challenges. Existing Semantic Web reasoners and Knowledge Base Management Systems (KBMS), primarily designed for powerful computing environments like servers or high-end mobile devices, are ill-suited for deployment on nano-scale devices. Reasoning engines that could potentially operate on smaller devices often lack support for crucial inference services, hindering their usefulness. Moreover, developing SWoE systems requires rethinking evaluation and benchmarking frameworks to account for the peculiar constraints of the novel paradigm.

This dissertation presents the work devoted towards realizing the SWoE vision, encompassing architectural designs and optimization approaches for several key elements of a complete toolchain, including: *Cowl*, a lightweight knowledge representation library tailored for resource-constrained devices, addressing the limitations of existing KBMS in the context of embedded and IoT devices, while remaining versatile enough to be useful at other scales of computation; *Tiny-ME*, a novel multi-platform reasoner and matchmaking engine for the SWoE, offering efficient reasoning capabilities suitable for cloud, desktop, mobile, and edge devices; *evOWLuator*, a cross-platform, energy-aware evaluation framework for Semantic Web reasoners, with a focus on power consumption estimation and support for inferences on remote devices, filling critical gaps in existing evaluation tools.

Strong emphasis is placed on the evaluation of the developed technologies through comprehensive experimental campaigns, whose results provide insights on performance, efficiency, and applicability in typical SWoE settings. Additionally, practical applications are demonstrated through case studies in diverse contexts. The first presented case study showcases how a smart city environment can be semantically enhanced using Cowl, demonstrating the SWoE's impact in city management by enabling nano-devices to exchange semantically augmented data. In a second scenario, Tiny-ME has been deployed to an unmanned aerial vehicle, facilitating autonomous, real-time decision-making for reliable drone operations. A third application, focused on supporting patients through semantic reasoning on wearable devices, shows how Tiny-ME can be used for real-time, explainable inferences on wearables

in highly dependable settings. Finally, a client-side Web reasoning use case emphasizes user privacy while granting efficiency and flexibility of personalized resource discovery. Collectively, the discussed evaluations and applications highlight the versatility and wide applicability of the proposed methods and technologies, underscoring the broad potential of the SWoE.

Contents

Introduction

In an increasingly connected world, the intricate network of the *Internet of Things* (IoT) weaves through the fabric of industry, environments, and daily lives. This dense mesh, where devices communicate and interact with each other over the Internet, has marked a significant stride in technological advancement. The IoT influences sectors ranging from home and building automation to industrial operations and environmental monitoring. It stands as a testament to the evolution of connectivity and computing devices, enhancing efficiency and convenience as well as enabling large-scale data collection and insightful analytics, thereby reshaping interactions between a spectrum of human activities and the physical world.

The integration of people, processes, and data in the communication fabric, along with the rapid proliferation and miniaturization of IoT technology, is ushering in the *Internet of Everything* (IoE), where living beings, objects, places and processes are interconnected and generate data streams, thus creating new opportunities in a wide range of economic and societal domains [32]. This transition exacerbates some of the well-known IoT challenges, such as network bandwidth usage and power consumption, and poses new concerns of its own, especially with respect to data protection and privacy.

The *edge computing* [110] paradigm attempts to mitigate some of these issues by moving computation and storage closer to where data are produced and used, namely towards the *edge* of the network, thus improving response time, saving bandwidth, and enabling the shift of ownership of collected data from service providers to end-users. A transition towards the edge is underway in the *Semantic Web* [13], which is shifting into the *Semantic Web of Things* (SWoT) [106]: under the SWoT vision, ontology-based annotations are associated to devices, objects, and phenomena to describe them in a

rich, structured, and unambiguous way. These metadata can be processed by intelligent agents via automated reasoning procedures to infer implicit knowledge, allowing them to act autonomously in the pursuit of their specific goals.

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, ranging from the *World Wide Web* (WWW) to nanodevices with very strict processing, memory and energy constraints. This vision requires supporting Knowledge Representation languages and automated inference on tiny devices, enabling autonomous decision-making, self-coordination and self-management, while also providing 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. In fact, the 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.

Achieving the SWoE vision requires building a practical software infrastructure that is aligned with its peculiarities and constraints, which turns out to be a pretty daunting task. Implementing such an architecture using existing technology, if at all possible, leads to sub-optimal results for a variety of reasons:

- Existing Semantic Web reasoners are mostly oriented to the WWW and expected to run server-side, or at least on powerful mobile devices such as high-end tablets and smartphones. As such, their deployment to nano-scale devices is usually unfeasible, due to technological and performance constraints. As will be shown in Chapter 3, very few systems are currently starting to meet SWoE requirements, though their use is still impractical due to either hardware constraints or platform interoperability issues.

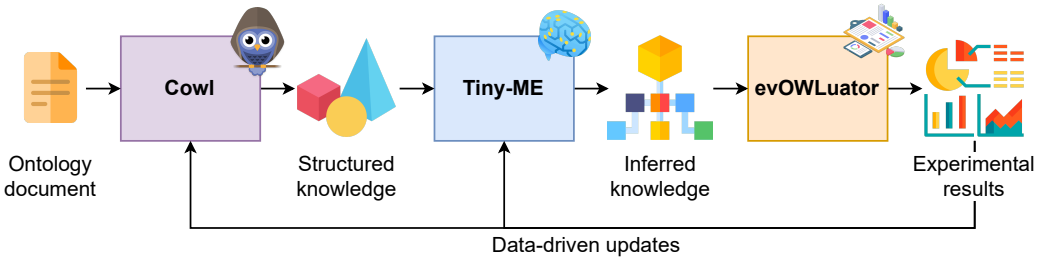


Figure 1: The proposed SWoE toolchain.

- As discussed in Chapter 2, similar considerations can be made for *Knowledge Base Management Systems* (KBMS), whose issues are aggravated by the dearth of libraries targeting the technological stacks of embedded and IoT devices.
- Reasoning engines that could theoretically be deployed to relatively small devices usually lack the support for non-standard inference services, which – as motivated in Section 1.2 – are crucially useful in the SWoT and, consequently, the SWoE.

Moreover, porting existing systems or designing new ones specifically for the SWoE requires rigorous evaluation methodologies, benchmarks, and software frameworks, as computational resource usage tends to become one of the primary limiting factors. The deployment of Semantic Web technologies to mobile and embedded devices also introduces concerns regarding energy usage, which has been historically disregarded in desktop-oriented systems. As will be detailed in Chapter 4, existing evaluation tools for Semantic Web technologies lack crucial features to be useful in the SWoE, such as power consumption estimation and support for running inference tasks on remote devices.

This dissertation summarizes findings of the Ph.D. program work conducted towards the achievement of the SWoE vision. The main academic results concern the following contributions, which make up the toolchain illustrated in Figure 1, hereby discussed in both their design and implementation:

- **Cowl** [14]: a lightweight knowledge representation library, specifically

tailored for resource-constrained devices.

- **Tiny-ME** [94][71]: a novel multi-platform reasoner and matchmaking engine for the SWoE.
- **evOWLuator** [104]: a cross-platform, energy-aware evaluation tool for Semantic Web reasoners able to run inference tasks on remote devices.

The remainder of the dissertation is organized as follows: Chapter 1 introduces the research context and recalls its technological background; Chapter 2 considers the requirements of a knowledge representation stack that is suitable for the SWoE, and discusses the design and implementation of the Cowl library; Chapter 3 presents the theoretical grounding behind efficient reasoning for devices in edge computing, and discusses the architecture and optimizations of the Tiny-ME reasoner; Chapter 4 introduces evOWLuator and demonstrates its effectiveness by reporting the results of a comparative evaluation campaign involving multiple state-of-the-art Semantic Web reasoners; Chapter 5 describes practical applications and case studies that validate the proposal, before conclusion and discussion about future perspectives.

Chapter 1

From the Semantic Web to the Semantic Web of Everything

1.1 The Semantic Web

The *Semantic Web* [13], often envisioned as the next evolution of the World Wide Web, is a collaborative movement led by the *World Wide Web Consortium* (W3C). It aims to transform the Web into a universal medium for the exchange of data, information, and knowledge, enabling machines to understand the semantics, *i.e.*, the meaning, of information on the WWW, thus making it a network of machine-interpretable *Linked Data* [47], in a way that is meaningful and useful for automated processes and applications.

To achieve this, the W3C has overseen the specification of a stack of technologies and standards. Cornerstones include: the *Resource Description Framework* (RDF) [103], which provides a framework for describing resources on the Web in a structured way; the *Web Ontology Language* (OWL) [85], which allows the creation of complex vocabularies for interpreting the meaning of terms and relationships within data; and the *SPARQL Protocol and RDF Query Language* (SPARQL) [122], a powerful query language to retrieve and manipulate information in RDF format.

The goal of the Semantic Web is not creating a new Web, but rather enhancing the existing one by providing *meaning* to Web resources through

semantically rich metadata, thus extending the Web of (human-oriented) documents with a Web of (machine-oriented) data that can be processed directly by automated agents, enabling them to support users more efficiently or execute tasks on their behalf. This includes improved search engines, advanced data integration, and more personalized user experiences, as machines can understand and respond to complex human requests based on the semantic understanding of digital information.

Central to realizing the Semantic Web's potential are two fundamental disciplines of the *Artificial Intelligence* (AI) field: *Knowledge Representation* (KR) and *Automated Reasoning* [4]. Knowledge Representation is concerned with creating a structured model of information, where data is not only stored but also linked to other relevant data in a way that defines their context and mutual relationships. This is where RDF and OWL play crucial roles, as they allow for the explicit representation of the semantics of data, making it possible for machines to understand and reason about the relationships between different pieces of information.

Automated reasoning, on the other hand, refers to the ability of computers to interpret the information and make logical deductions based on it. This is crucial for the Semantic Web, as it enables machines to perform complex tasks such as intelligent search, data integration, and even decision-making based on the understanding of data semantics. With automated reasoning, computer agents can infer new information from existing data, identify inconsistencies, and provide more accurate responses to complex queries.

Together, *Knowledge Representation and Reasoning* (KRR) techniques are essential for turning the WWW into a more intelligent and responsive environment, enabling a new level of interaction where machines can effectively process, analyze, and even anticipate user needs based on the rich semantic structure of annotated Web resources.

1.1.1 Knowledge representation in the Semantic Web

An *ontology* is an *"explicit specification of a conceptualization"* [42], *i.e.*, a structured and machine-understandable representation of a domain, encompassing its key concepts, relationships, and constraints. It defines a shared vocabulary for users and user agents, and enables reasoning about knowledge concerning the domain [114]. The OWL [85] W3C standard ontology language for the Semantic Web is formally grounded in *Description Logics* (DLs) [4], a family of knowledge representation languages that provides the underlying semantics to represent knowledge and reason about it. DLs are decidable fragments of *First Order Logic* (FOL) [20, 33] which allow the formal representation of knowledge by means of:

- *concepts*, representing sets of objects of the domain;
- *individuals*, instances of concepts, *i.e.*, actual objects in the domain;
- *roles*, defining relationships between pairs of individuals.

These elements can be combined via *constructors* to create DL expressions, whose formal semantics is specified by means of an *interpretation* $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$ associating each term to a subset of the universe of discourse (the *domain* Δ) by means of an *interpretation function* $\cdot^{\mathcal{I}}$. Concept *conjunction* is interpreted as set intersection: $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$. Concept *disjunction* is interpreted as set union: $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$. The connector $\neg$, if present, is interpreted as *complement*. More constructs exist, which distinguish each language of the DL family, determining their expressiveness and the computational complexity of inference tasks.

In the context of DLs, an ontology (a.k.a. terminology, terminological box, TBox) [114] is composed of two types of *assertions* involving concepts and roles: *inclusion*, which allows the definition of *is-a* relationships between concepts ($A \sqsubseteq D$ where A and D are concept expressions); *equivalence*, which allows naming expressions, or specifying that two expressions represent the

same set of instances ($A \equiv D$). Individuals, and relationships between them, make up the so-called *assertion box* (ABox). A Knowledge Base (KB) consists of a $\langle$TBox, ABox$\rangle$ pair. With respect to OWL, oftentimes the term "ontology" is informally used to refer to a KB as whole, rather than just its terminological part.

The W3C published official Recommendations for two versions of OWL, in 2009 and late 2012, respectively. OWL 2 built upon the foundational principles of the initial version by increasing the language expressiveness, and extending datatype support and annotation capabilities.

Entities are the essential building blocks of OWL 2 ontologies and serve as the foundation for defining the vocabulary of an ontology, consisting of named terms. Each entity is uniquely identified by an *Internationalized Resource Identifier* (IRI) [34], an extension of the *Uniform Resource Identifier* (URI) [12] with a larger alphabet of allowed characters. Entities encompass the following categories:

- **Classes**: sets of individuals, corresponding to a DL concept. For instance, the class *Dog* might include individuals like *Scooby-Doo* and *Snoopy*. Classes can have hierarchies, where one class is a subclass of another. For example, *Dog* can be a subclass of *Mammal*, meaning every dog is also a mammal.

- **Individuals**: specific instances or objects that belong to classes. For instance, *Shaggy* can be an individual of the class *Person*.

- **Object Properties**: relationships between individuals, corresponding to DL roles. For instance, the property *friendOf* can relate the individual *Scooby-Doo* to the individual *Shaggy*, indicating that Scooby-Doo is Shaggy's friend.

- **Datatype Properties**: assign data values to individuals, corresponding to DL functional roles on concrete domains. For instance, an *age* property might assign the value *17* to the individual *Shaggy*.

- **Annotation Properties**: encode information about parts of the ontology itself, rather than the domain of interest. For example, they might provide metadata about when an axiom was added or who the author of a particular part of the ontology is.

Additionally, OWL allows defining *anonymous individuals*, which are useful for representing information about something without specifying a unique identifier. For instance, if one wants to state that there exists a person who is an enemy of *Shaggy* but does not want to specify who that person is, they might use an anonymous individual to represent that person.

Just as in DLs, OWL entities can be combined in more complex *expressions* by using logical *constructors*, characterizing sets of individuals by precisely outlining criteria related to their properties. Individuals that fulfill these criteria are deemed instances of the corresponding class expressions. For instance, atomic classes like *Dog* and *Cartoon* can be conjunctively combined to describe the class of cartoon dogs. Every OWL 2 ontology is a collection of *axioms*, basic statements that assert what is true in the domain of interest by combining entities and expressions. OWL allows for a variety of axiom types, allowing the composition of entities and expressions into many types of logical assertions.

The variety of logical constructors supported by OWL makes it highly expressive, allowing the specification of intricate domain knowledge, but also posing challenges for the computability and implementation of inference tasks. To address this, OWL 2 introduced several *profiles*, streamlined versions of the language tailored to meet specific industrially relevant use cases, such as enhanced reasoning scalability and efficient query answering:

- *OWL 2 EL* is optimized for ontologies with large numbers of properties or classes. It is particularly suited for fields like life sciences, due to its support for extensive hierarchies.
- *OWL 2 QL* is designed for applications requiring efficient access to large amounts of instance data. It facilitates easy integration with relational

databases, making it ideal for applications where ontology-based data access is critical.

- *OWL 2 RL* is targeted at applications that require scalable reasoning without sacrificing too much expressiveness. It is particularly useful in scenarios where reasoning can be implemented using rule-based systems.

Each profile strikes a balance between expressiveness and performance, allowing practicioners to choose the most appropriate subset for their specific use case, ensuring that OWL remains versatile and applicable across a wide range of domains. In any case, knowledge-based systems are not required to fully conform to any specific OWL profile: supported OWL constructs may be arbitrarily restricted, so that they align with DLs that are proven to have favorable computational complexity and practical performance characteristics, while still being sufficiently expressive for useful applications. One such DL is the *Attributive Language with unqualified Number restrictions* ($\mathcal{ALN}$), which provides moderate expressiveness while keeping polynomial space and time complexity for inferences [30]. Since $\mathcal{ALN}$ will be the reference expressiveness for most of this dissertation, its constructors are reported in what follows and Table 1.1 summarizes the syntax and semantics of its constructors and assertions.

- $\top$, *universal concept.* All the objects in the domain.
- $\bot$, *bottom concept.* The empty set.
- A, *atomic concepts.* All the objects belonging to the set A.
- $\neg A$, *atomic negation.* All the objects not belonging to the set A.
- $C \sqsubseteq D$, *intersection.* The objects belonging to both C and D.
- $\forall R.C$, *universal restriction.* The objects x such that if x is related to y by the relation R, then y belongs to the set C.
- $\exists R$, *unqualified existential restriction.* The objects that are related to at least one object by the relation R.

- $\geq nR$, $\leq nR$, $= nR$, *unqualified number restrictions*[1]. The objects that are related to at least, at most, or exactly n objects by the relation R.

Table 1.1: Syntax and semantics of $\mathcal{ALN}$

Name	Syntax	Semantics
Top	$\top$	$\Delta^{\mathcal{I}}$
Bottom	$\bot$	$\emptyset$
Intersection	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Atomic negation	$\neg A$	$\Delta^{\mathcal{I}} \backslash A^{\mathcal{I}}$
Universal quantification	$\forall R.C$	$\{d_1 \mid \forall d_2 : (d_1, d_2) \in R^{\mathcal{I}} \rightarrow \ d_2 \in C^{\mathcal{I}}\}$
Number restrictions	$\geq nR$	$\{d_1 \mid \sharp\{d_2 \mid (d_1, d_2) \in R^{\mathcal{I}}\} \geq n\}$
	$\leq nR$	$\{d_1 \mid \sharp\{d_2 \mid (d_1, d_2) \in R^{\mathcal{I}}\} \leq n\}$
Inclusion	$A \sqsubseteq D$	$A^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
Equivalence	$A \equiv D$	$A^{\mathcal{I}} = D^{\mathcal{I}}$

In practical usage, a specific syntax is required for storing and exchanging ontology documents across tools and applications. Ontologies can be serialized in any triple-based RDF syntax, such as *RDF/XML* [103] (RDF encapsulated in *eXtensible Markup Language*) or *Turtle* [90]. Further OWL-specific syntaxes, such as *Functional* [85] and *OWL/XML* [77], directly encode OWL axioms without translating them into the corresponding RDF triples. Finally, the *Manchester syntax* [50] is designed to be more readable for users without a background in logic. Tools are available for translating between different syntaxes, ensuring flexibility and adaptability for various use cases and user preferences.

1.1.2 Automated reasoning in the Semantic Web

The close relationship with DLs allows OWL ontologies to capture complex domain knowledge, supporting automated reasoning to infer new knowledge

[1]It is useful to notice that $\exists R$ is equivalent to $\geq 1R$ and that $= nR$ is a shortcut for $\geq nR \sqcap \leq nR$.

based on explicitly defined facts and relationships. As a result, tools and systems built on OWL can make sophisticated deductions and answer complex queries by leveraging the foundational principles of DLs. Automated reasoning algorithms usually fall in one of the following categories:

- *Structural algorithms* focus on the structure of logical expressions. They work by analyzing and manipulating the form of the expressions, often transforming them into simpler or more standardized forms. These algorithms are particularly useful in contexts where the logical structure itself is complex or plays a crucial role in the reasoning process, and they are usually very efficient (polynomial). Their disadvantage is that they are incomplete, *i.e.*, they cannot detect all the existing relationships, for very expressive DLs.

- *Tableaux-based algorithms* work by constructing a *tableau*, a tree-like structure representing various interpretations of the ontology. By expanding this tableau according to specific rules and checking for contradictions, these algorithms can determine, as an example, if an ontology is consistent or if a certain concept expression is satisfiable. This family of algorithms can handle much more expressive DLs, albeit at a significantly greater (exponential, and sometimes double exponential) time and space complexity.

- *Rule-based algorithms* operate by applying sets of predefined rules, typically in the *antecedent-consequent* form, to a set of asserted facts, generating new (inferred) facts. In *forward chaining*, rules are applied iteratively to the growing set of facts, until some specific fact is derived (*e.g.*, if the system is asked to answer a query) or until no more facts can be derived. Similarly, *backward chaining* starts with the query and works backwards, applying rules to determine which facts must be true to satisfy it. Rule-based algorithms are usually efficient, requiring polynomial time in the size of the input, however they are rather rigid and do not allow *non-monotonic* reasoning, *i.e.*, whose conclusions can be retracted based on further contradicting facts.

With regard to the $\mathcal{ALN}$ DL, complete structural inference algorithms are well known both for standard ([4], 2.3.1) and non-standard, non-monotonic inferences [93]. They are based on the concept unfolding and Conjunctive Normal Form (CNF) normalization preprocessing steps. Basically, **concept unfolding** recursively expands terminological axioms in the TBox 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, $\mathcal{ALN}$ TBoxes are subject to the following limitations [93]:

- the left-hand side (LHS) of inclusion ($\sqsubseteq$) and definition ($\equiv$) axioms must be atomic, *i.e.*, *general concept inclusions* are not allowed;
- if an atomic concept A is the LHS of a definition axiom, then it cannot be the LHS of any other inclusion or definition axiom;
- TBoxes must be *acyclic*, *i.e.*, the concept at the LHS of inclusion and definition axioms must not appear in the unfolding of the right-hand side (RHS) of the same axiom. This requirement can be partly relaxed, by allowing *told subsumption cycles* [124].

CNF normalization translates the unfolded concept expression in a canonical form that preserves its semantics. In $\mathcal{ALN}$ CNF, every concept expression is either $\bot$ 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.

Given a DL ontology $\mathcal{T}$ and S, R two concepts in $\mathcal{T}$, the *satisfiability* and *subsumption* standard inference services provided by DL-based systems [4] can be formalized as follows:

- **Subsumption**: checks if S is more specific than R w.r.t. the ontology $\mathcal{T}$, *i.e.*, $\mathcal{T} \models S \sqsubseteq R$. In this case, all instances of S are also instances of R.

- **Satisfiability**: checks if S can have instances w.r.t. the ontology $\mathcal{T}$, *i.e.*, $\mathcal{T} \not\models S \sqsubseteq \bot$. An unsatisfiable class contains a contradiction, which implies that it cannot have any instance.

Once $\mathcal{ALN}$ concept expressions have been unfolded and normalized, subsumption and satisfiability can be carried out by looking at the structure of the expressions, comparing them as if they are sets of primitive atoms representing the $\mathcal{ALN}$ constructs in Table 1.1. The algorithmic implementation of the above services, along with illustrative examples of their application on a toy ontology, are reported in Section 3.2.

General knowledge-based applications usually require additional, more complex inference services over ontologies, such as *ontology coherence*, *consistency*, and *classification*:

- **Ontology Coherence** involves checking that all named concepts in the TBox are satisfiable [76]. If the ontology has an unsatisfiable class, then it is incoherent, though useful inferences can still be drawn from it. As such, incoherent ontologies can be and are often used in applications.

- **Ontology Consistency** determines whether it is possible to interpret the axioms in the ontology such that there is at least one class which has an instance. If the ontology is inconsistent, every class is interpreted as the empty set, and as such no useful conclusions can be drawn. This is generally regarded as a severe error in ontology modeling, and most reasoners just abort inferences when faced with this condition.

- **Ontology Classification** computes the overall concept taxonomy induced by the subsumption relation, from $\top$ to $\bot$. Essentially, classification is logically equivalent to computing all the subsumption relations between all pairs of concepts in the TBox. As such, it is a rather

complex reasoning task, which requires careful optimization in order to be completed in reasonable time and space over large ontologies.

1.2 The Semantic Web of Things

The *Semantic Web of Things* (SWoT) [106] is an emerging concept that combines the Semantic Web's goal of creating a more meaningful and usable web with the *Internet of Things* (IoT), which connects everyday devices to the Internet. The fundamental idea is to apply semantic technologies to the IoT, thereby facilitating better data integration, interpretation, and usability across various interconnected devices.

In the SWoT, ontology-based annotations are used to describe devices, objects, and phenomena in a detailed and standardized manner. These semantically rich metadata become the language through which intelligent agents understand and interact with the world. By leveraging automated reasoning procedures, agents can infer implicit knowledge from the explicit descriptions provided, and act autonomously in the pursuit of their specific goals without requiring constant human oversight or intervention. This aspect of the SWoT opens up possibilities for more responsive, adaptive, and intelligent IoT systems.

A critical distinction between the SWoT and the traditional Semantic Web lies in the nature and scale of the KBs they operate upon and the queries they handle. In the conventional Semantic Web, the focus is often on complex queries across large KBs that exhibit moderate-to-high expressiveness. These inferences are typically run as intensive batch jobs, requiring significant processing power and time. However, the SWoT is highly heterogeneous and characterized by the need to cater to a wide range of specialized use cases, which often involve smaller KBs with low-to-moderate expressiveness. The nature of inferences in the SWoT is also markedly different, emphasizing quick, on-the-fly queries to respond to rapidly changing conditions and provide insights or enable actions immediately.

This paradigm shift is pivotal in making the SWoT practical and effective in real-world applications. By focusing on smaller, more agile KBs and rapid queries, the SWoT can be more effectively integrated into a variety of environments, from smart homes and cities to industrial settings and beyond. This approach allows for a more dynamic interaction between the semantic layer and the physical world, enabling IoT devices to not only collect and exchange data but also to understand and respond to it in meaningful ways.

1.2.1 Ubiquitous knowledge bases

The *ubiquitous Knowledge Base* (u-KB), originally introduced in [97], is a cornerstone in realizing the Semantic Web of Things, standing as the architectural model that underpins the integration of the Semantic Web and the IoT. In this framework, physical objects imbued with semantic-enabled capabilities can be discovered, queried and inventoried in a peer-to-peer, collaborative way, without requiring any central control and coordination. The u-KB provides a comprehensive architectural blueprint that connects mobile ad-hoc networks used in ubiquitous computing with the Internet. It also incorporates a distributed application-layer protocol that operates on a peer-to-peer basis, enabling the dissemination and discovery of knowledge. Various identification and sensing technologies are employed to collect information, which is then used by inference engines and semantic-aware applications through a uniform set of operations, in pervasive environments as well as in the Web.

The essence of u-KB lies in its ability to handle the challenges posed by the dynamic and decentralized nature of the SWoT environments, addressing the need for efficient information storage, management, and discovery, and offering transparent access to information sources within a given area. It adopts Semantic Web languages and technologies, allowing for a detailed annotation and categorization of resources, thereby empowering semantic-based applications to leverage tools for querying, reasoning, and matchmaking, all underpinned by formal logic principles inherited by the Semantic Web initiative. This

semantic enrichment is crucial for enabling nuanced machine-to-machine communication and supporting the automation required in dynamic IoT environments.

Central to the framework's design is the *content-centric* networking approach: instead of addressing the physical locations of data storage, this technique focuses on the content itself, aligning with the fluid nature of IoT where devices are often mobile or transient. This shift is essential for the network's adaptability, ensuring that content can be retrieved irrespective of the changing states or positions of devices. The framework is also designed to be fully decentralized, with a two-level infrastructure that includes a *field layer* for connecting embedded micro-devices, and a *discovery layer* for inter-host communication. Adoption of suitable peer-to-peer protocols enhances the system's decentralization, resilience, and scalability, allowing devices to discover and query information autonomously, without relying on a centralized authority.

Semantic query languages and reasoning tools embedded within the u-KB framework enable sophisticated resource discovery and matchmaking, thus allowing the system to go beyond simple data retrieval: user queries and device data are interpreted according to a specific context, finding and linking information that is most pertinent to the user's current needs. A peculiarity of the framework is its support for autonomy and collaboration among devices. Through shared vocabularies and ontologies, devices participating in a u-KB can independently register, deregister, and identify other devices and services, while ensuring that they can communicate effectively, by sharing and interpreting information in a unified language.

The u-KB framework adopts a multi-layered architecture designed to support semantic information dissemination and resource discovery in pervasive environments. The layers of the framework, sketched in Figure 1.1 and outlined in what follows, interact to orchestrate the information flow from the physical world to the application layer, allowing for intelligent decision-making and data processing.

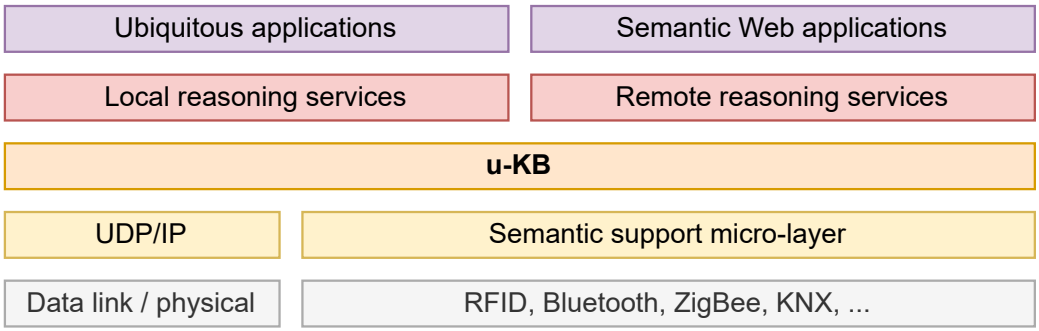


Figure 1.1: Ubiquitous Knowledge Base framework.

- **Data Link/Physical Layer:** includes *field layer* protocols and technologies that enable the physical interconnection of embedded devices in the environment, such as RFID [26], Bluetooth [17], ZigBee [27], KNX [60], and others. A *discovery layer* handles the discovery of resources and services, allowing each network host to act as a cluster head for field devices in its direct range using available communication interfaces.

- **Semantic Support Micro-Layer:** at the base of the framework, this layer ensures interoperability with mobile ad-hoc networks of embedded devices and sensors. It adapts the mobile identification and sensing technologies of the physical layer to the semantic requirements of the framework.

- **u-KB Layer:** provides common access to information from semantically enhanced devices and sensors populating a smart environment. It uses the Internet Protocol (IP) for basic addressing and routing in local networks and between autonomous networks, including wide area networks and the Internet.

- **Local and remote reasoning services:** provided to both local pervasive applications and remote Semantic Web applications. They enable logic-based queries and reasoning capabilities that are crucial for processing semantically annotated information.

This layered approach allows for a robust, flexible, and scalable system that can handle the dynamic and diverse requirements of the Semantic Web

of Things. The framework supports semantic-aware applications in mobile ubiquitous contexts as well as in the Web, ensuring that information gathered through different identification and sensing technologies can be effectively exploited, both in ubiquitous and classical Semantic Web applications.

1.2.2 Micro-reasoners and non-standard inferences

The standard inference services introduced in Section 1.1.2, while useful in traditional semantic-enabled applications, are usually not enough in scenarios that require more than just a Boolean answer, such as in matchmaking or negotiation, which are customary in the SWoT and, consequently, the SWoE. In those cases, non-standard, *non-monotonic* inferences [93] are more useful, as they provide explanations for inference outcomes, and allow retracting conflicting information in the Open World Assumption. Non-standard inference services are formally introduced in what follows, while their algorithmic implementation and some illustrative examples are reported in Section 3.2.

Semantic matchmaking is defined as the problem of finding the most relevant element in a set of *resources* given a *request*, where both request and resources are represented as satisfiable concept expressions with respect to a common set of axioms $\mathcal{T}$ in a DL. The output of semantic matchmaking consists of a set of concepts, each with a score representing its semantic relevance to the submitted query. Besides resource discovery, matchmaking can support data stream analytics, allowing the conversion of statistical classification problems into semantic matchmaking ones. As an example, in [100] 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 [28, 95, 138]. The present work refers to the theoretical framework grounded on the classification of match types proposed in [30]. Similar classifications were suggested in [66] and [83]. Differing from those earlier works, however, the framework considered in this dissertation has a different order of preference

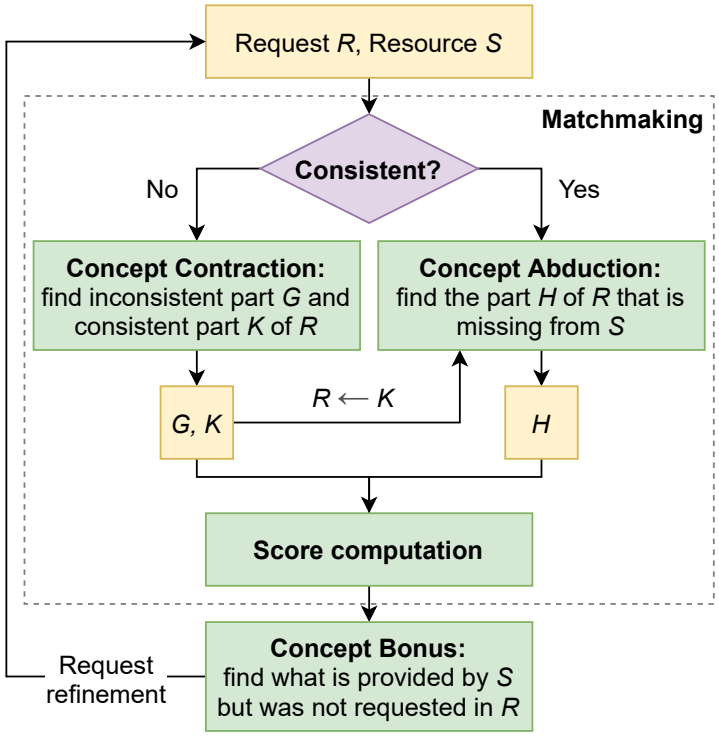


Figure 1.2: Semantic matchmaking framework.

among the various non-exact match category, with *full/subsume* being the favorite one due to not needing to hypothesize additional information about the resource, as explained in what follows. Furthermore, the adopted framework enables ranking different resources within the same match category. More recent works [93, 87] are oriented to resource discovery in the SWoT and combine reasoning and quantitative contextual attributes in *utility functions*.

The *semantic matchmaking* framework sketched in the flow chart of Figure 1.2 combines the standard and non-standard inference services outlined hereafter. Let us consider a set of axioms $\mathcal{T}$ in $\mathcal{ALN}$, and R, S two concepts in $\mathcal{L}$ –representing a *request* and a *resource*, respectively– both satisfiable in $\mathcal{T}$. Semantic matchmaking requires a preliminary **Consistency check** 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* scenario.

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 $H \in \mathcal{L}$ 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 are defined under the Open World Assumption [93], *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* [66, 83]), 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.[2]

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 resource provides even though the request did not ask for it.

The $\mathcal{ALN}$ CNF for concept expressions induces the definition of a metric space with a *norm* operator $\|\cdot\|$. In the above matchmaking framework, 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 [93] penalty values have

[2] *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.

been proposed where:

1. each (possibly negated) atomic concept counts as 1;
2. for each role, number restrictions are weighted as the relative difference between cardinality values in R and S w.r.t. the cardinality value in R (if R lacks a number restriction on that role, the penalty is 1);
3. for each value restriction, the penalty is computed recursively as above.

For CA, CB, and CC, 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 [120], if $\mathcal{T} \models R \sqsubseteq S$, then the output of difference $D = CD(R, S)$ is a concept $D \in \mathcal{L}$ such that $R \equiv S \sqcap D$. In that case, one typically wants to subtract as much as possible.

While CA, CB, CC, and CD are useful in one-to-one discovery, matchmaking and negotiation scenarios, the **Concept Covering** (CCov) non-standard inference can be exploited to compose a set of elementary expressions to answer complex requests [107]. 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 $\mathcal{S} = \{S_1, S_2, ..., S_n\}$ in a KB (available resources), where R and S_1, S_2, ..., 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 $\mathcal{S}$ 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 $\mathcal{S}_c$.

1.2.3 Open issues

By joining the Semantic Web vision of a meaningful and usable Web with the ubiquitous networking capabilities of the IoT, the SWoT has marked a significant stride in semantic-enabled digital ecosystems. However, as the SWoT progresses towards the more pervasive and encompassing SWoE, a new set of issues arises. The transition is not merely an expansion in scale or scope, but involves a fundamental shift in the deployment and application of KRR technologies. The core of the challenge lies in the need to implement them on an even more granular level, moving from mobile devices to microcontrollers and nano-scale devices that form the very fabric of the IoE.

This leap forward is fraught with complexities, as the constraints of these smaller devices in terms of computational power, memory, and energy are significantly more stringent. The challenging task of embedding semantic capabilities into such constrained environments, while retaining compatibility with conventional Semantic Web and SWoT contexts, requires novel solutions and a rethinking of existing paradigms. Moreover, the inherent volatility of devices, the high velocity of data streams, platform heterogeneity, and the intrinsic need for cross-platform interoperability further complicate this transition. As such, the journey from the SWoT to the SWoE exposes a multitude of open issues, demanding a concerted effort to develop robust, efficient, and scalable solutions that can support novel machine-to-machine and human-to-machine interaction models, applications and services to really realize the vision to it full potential.

One of the primary challenges of the IoE is the high **volatility** of devices and network links. Devices in typical IoE ecosystems are often mobile, leading to dynamic network topologies that constantly change, and intermittent availability of Internet access. In this setting, relying on powerful external devices to provide semantic support is not always feasible, and thus KB manipulation and automated inference primitives must be locally available. Additionally, the reliance on battery power for many of these devices introduces constraints on their operational lifetime and performance. Addressing these issues re-

quires the development of lightweight, energy-efficient KRR technologies that can adapt to the highly dynamic nature of SWoE environments.

Velocity of the generated data is a significant hurdle for traditional Semantic Web systems, especially in real-time processing settings within the IoT. As recalled previously, their focus is usually on large and expressive KBs, and highly sophisticated and costly boolean inferences, which are often incompatible with applications involving high-velocity data streams. SWoE systems must instead emphasize quick queries on small and moderately expressive KBs, possibly through more versatile non-monotonic inferences, to respond to rapidly evolving conditions with immediate insights and actions.

The SWoE is also characterized by a high degree of **platform heterogeneity**, stemming from the vast array of hardware configurations and software ecosystems present in the IoE landscape, where devices range from powerful servers to the most constrained microcontrollers, each with its own set of capabilities, operating systems, and communication protocols. While several Java-based Semantic Web reasoners have been ported to Android [18], the landscape is almost completely barren for iOS devices [98] and embedded real-time operating systems. A practical SWoE infrastructure requires new cross-platform inference frameworks, services and tools that are capable of functioning effectively on high-capacity workstations and mobile devices, as well as being sufficiently lightweight to run on nano-devices with limited computational resources.

The **resources constraints** of the lower end of the SWoE device spectrum, in fact, pose significant constraints on the deployment of traditional Semantic Web technologies. Most existing Semantic Web reasoners and KBMS are designed for the WWW, with expectations of running on plugged-in servers or at least on capable mobile devices like tablets and smartphones. Porting existing tools is often unfeasible because of deep hardware and software platform differences, and even when possible it is a major effort which does not always pay off in terms of performance. Specifically, energy consumption has seldom been considered in reasoner design for conventional computing architectures. There is a pressing need for the development of lightweight,

energy-efficient semantic technologies specifically tailored for the SWoE.

Finally, to really find out whether new or adapted solutions are really fit for the SWoE, rigorous evaluation methodologies and frameworks are needed. As will be shown in Chapter 4, current OWL reasoner evaluation frameworks and tools do not provide a fully satisfactory solution in terms of flexible support of both standard and non-standard reasoning services, wide platform compatibility, energy consumption estimation, and ability to run inference tasks on remote devices, in a single package.

Chapter 2

Cowl: knowledge representation from nano to Web scale

This chapter introduces *Cowl* [14], an innovative OWL manipulation library tailored for handling ontologies on a wide variety of platforms, from workstations and mobile devices to embedded systems with stringent processing and storage constraints. Its source code[1] is openly available under a very permissive license[2], seeking to encourage adoption in both academia and industry. The library supports parsing, querying, editing, and serializing OWL ontologies, and focuses on processing efficiency and low memory usage by means of targeted design choices, features, and optimizations. The toolkit also introduces the *axiom streams* technique, a novel approach to ontology processing at the axiom level which does not rely on an intermediate data store, optimizing resource usage and aligning with settings characterized by tight resource limitations. A comprehensive evaluation campaign on a popular embedded platform validates its usability on the smallest end of the SWoE device spectrum, constituting the first successful deployment and evaluation of a fully featured OWL library on a microcontroller with less than 100 KiB of RAM.

The remainder of the chapter is as follows: Section 2.1 discusses relevant related work and tools, while Section 2.2 and Section 2.3 describes capabilities

[1]Cowl source code: `https://github.com/sisinflab-swot/cowl`
[2]Eclipse Public License v2.0: `https://www.eclipse.org/legal/epl-2.0/`

and architecture of the Cowl library, respectively; Section 2.4 introduces the novel axiom stream technique for ontology parsing and serialization; Section 2.5 focuses on peculiarities and optimizations that enable its use on any device category; finally, experiments are reported and results discussed in Section 2.6.

2.1 Background

The available software tools designed for OWL manipulation generally rely on high-level abstractions, enabling applications to interact with ontologies without dealing with the details of the underlying serialization formats. Historically, the design of OWL software has been influenced by the the strong tie that exists between OWL and RDF. As such, the first OWL toolkits adopted a *triple-based* abstraction, where ontologies are seen as sets of RDF triples, and APIs are provided to help the user manipulate them consistently with the OWL specification.

One such tool is *Apache Jena* [75], a Java framework that provides manipulation primitives for RDF, *RDF Schema* (RDFS) [24] and OWL models, and an inference Application Programming Interface (API) to support reasoning and rule engines. Another example is the *Protégé-OWL Plugin* [59] for the *Protégé* ontology editor [78], which is built on top of Jena and is particularly effective for developing graphical applications. The *owlcpp* [65] C++ open-source library allows parsing OWL ontologies in RDF/XML format into an RDF triple store constructed on the *Redland* [9] application framework, further employing the *Fact++* [124] engine for reasoning tasks.

Tools grounded on the triple-based abstraction usually adopt RDF statements and triple stores as their core data models, and target RDF-based OWL serializations. This is problematic for a number of reasons: firstly, they often mandate varying levels of familiarity with the way OWL ontologies map to RDF graphs, making them more technical and challenging to use; secondly, they represent OWL axioms as one or more RDF triples, making updates

and queries more complex and usually introducing memory overhead; finally, RDF-based serializations mandate the presence of parsers and renderers for RDF syntaxes, and sometimes XML as well, increasing code size and memory requirements.

The *OWL API* [49], a widely adopted Java-based OWL toolkit, pioneered a different approach to OWL abstractions, where ontologies are seen as collections of axioms, and APIs are more closely aligned with the OWL specification. This *axiom-based* abstraction completely hides serialization details, effectively enabling the use of arbitrary memory storage techniques, and smoothening the learning curve for practitioners. The *OWL API for iOS* [96] is a partial Objective-C port of the OWL API targeted at Apple devices, which can be combined with the *Mini-ME Swift* reasoner [98] to enable semantic-enhanced applications and services on the iOS operating systems lineup. This approach is also adopted by *Horned-OWL* [70], an early-development Rust-based OWL 2 manipulation library aimed at large-scale ontologies, which provides full parsing for OWL-XML and nearly full parsing for OWL-RDF.

A further abstraction, the so-called *ontology-oriented programming* technique, integrates ontologies directly into a runtime environment, treating ontology entities like objects in a programming language. This strategy leverages the similarities between ontology structures and object-oriented programming (OOP) models to simplify the integration of ontologies and reasoning in software development by adopting a familiar syntax, though it often requires significant introspection support at the language level. *Owlready* [62], one of the first proponents of this paradigm, is a popular toolkit for manipulating OWL ontologies and their constructs as Python objects, as well as for reasoning over them via modified versions of the *HermiT* [38] and *Pellet* [113] engines. A similar approach is adopted by *OWLOOP* [25], an extension of the *OWL-API* that reduces the code required to manipulate ontologies by enabling interaction with OWL entities as objects within the OOP paradigm, though it currently only supports a subset of OWL 2 axioms.

Despite the abundance of OWL manipulation tools highlighted in this

Table 2.1: Comparison of OWL toolkits.

Toolkit	Language	Updated	Memory usage
Apache Jena [75]	Java	2023	High [11]
Protegé-OWL [59]	Java	2020	N.A.
owlcpp [65]	C++	2016	Medium [14]
OWL API [49]	Java	2023	High [14]
horned-owl [70]	Rust	2023	N.A.
OWL API for iOS [96]	Objective-C	2020	Medium [96]
Owlready2 [62]	Python	2023	High [14]
OWLOOP [25]	Java	2022	N.A.
Cowl	C	2023	Low [14]

short literature review, most of them exhibit significant issues that hinder their suitability for resource-constrained platforms, as outlined in Table 2.1, including:

- dependency on the Java and/or Python runtime environments, which are incompatible with embedded devices[3];
- designs tailored for conventional computing environments, assuming high availability of processing power and memory resources;
- lack of comprehensive performance evaluations, especially concerning memory usage and targeting resource-constrained devices;
- not actively maintained projects.

A preliminary version of the Cowl library was described in [14], where an early assessment indicated a promising alignment with the requirements of the Semantic Web of Everything. Subsequently, critical functionalities have been added, such as support for ontology editing and serialization, and

[3]The availability of Python interpreters designed for embedded systems like *MicroPython* [10] does not overcome this issue, as they include only a subset of standard Python runtime environment libraries, thus porting existing OWL manipulation tools would require at least a partial rewrite.

management of ontologies as continuous streams of axioms. Additionally, the tool has been thoroughly evaluated on an embedded platform.

2.2 Capabililties

Cowl is a complete implementation of the *OWL 2 Structural Specification* [85]. It supports OWL 2 ontologies with no restrictions on the structure of axioms and expressions. At a high level, the library provides the following features:

- parsing ontology documents;
- submitting programmatic ontology queries;
- creating and editing ontologies;
- serializing and writing ontology documents.

Ontology documents can be parsed into an optimized in-memory store, which can then be queried to retrieve knowledge. The store can be edited by adding and removing axioms, and then serialized to make changes persistent. Ontologies can also be read and written as *axiom streams*, a novel technique (cfr. Section 2.4) where the document is processed axiom-by-axiom without needing an intermediate store, which is particularly desirable for memory-constrained platforms.

Cowl currently supports ontology documents serialized in the *functional syntax* [85], though its architecture allows for multiple readers and writers which can be implemented in future updates. The functional syntax has been chosen as the reference format because it has a few useful characteristics:

- simple, unambiguous grammar which can be parsed with very little resources;

- no requirement for underlying serialization formats, such as RDF or XML, which would require dedicated readers and writers;
- guarantee that the byte representation of an axiom is never interleaved with that of any other axiom. As explained in what follows, this enables significant memory savings when processing OWL knowledge graphs as axiom streams.

2.3 Architecture

The foundational design choice concerns the selection of an appropriate programming language, which must certainly be a systems language so that the library can be deployed to embedded devices, but also interoperable enough to be easily integrated in other, higher-level languages and runtimes, granting support for mobile devices, desktops, and cloud servers. The C language is a sensible choice, as it is the most widely used language on embedded devices [46] and also among the most interoperable [16]. The library has thus been implemented in standard C11, with no compiler-dependent extensions or platform-specific API calls, allowing it to be effortlessly deployed to any platform equipped with a C compiler, and integrated into any programming environment that provides C interoperability.

This choice has ruled out the adoption of the *ontology-oriented* programming abstraction, as the C language lacks the required object-oriented features and introspection capabilities. The *triple-based* abstraction has also been discarded, as it would require additional software components and computational resources. The architecture of the Cowl library is sketched in Figure 2.1. It employs the *axiom-based* abstraction, and its data model and APIs are designed to closely follow the OWL 2 Structural Specification [85] W3C Recommendation. Mirroring the specification in C is a design challenge: relationships between OWL constructs are often hierarchical, and cannot be easily rendered with C types due to the purely procedural nature of the language, with no support for object-oriented features such as inheritance.

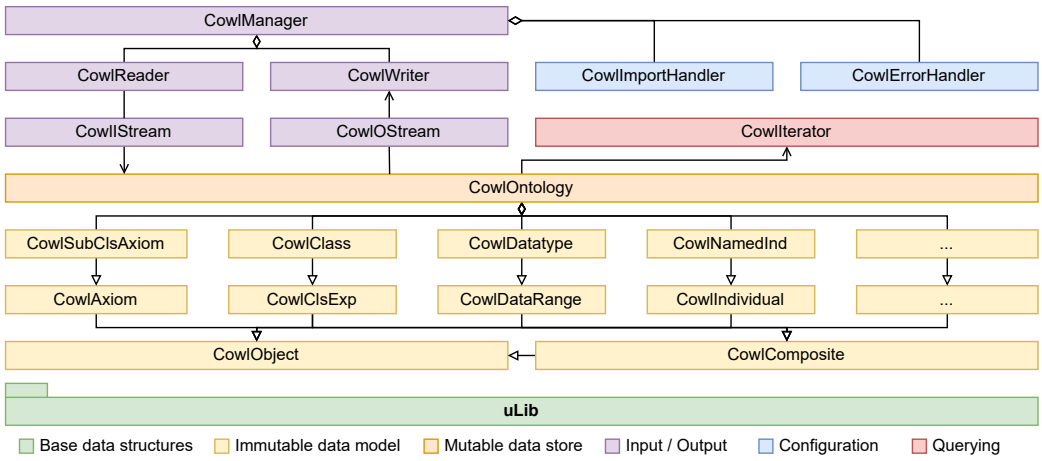


Figure 2.1: Architecture of the Cowl library.

In Cowl, hierarchical relationships are thus emulated through the use of the *pseudo-inheritance* technique, which exploits specific features of the C language to simulate inheritance at the structure level: the C11 specification ([52], §6.7.2.1-15) guarantees the absence of padding at the beginning of a *struct*, ensuring that its memory address coincides with that of its first member. By nesting *struct*s in a manner where the base type (the "superstruct") is the first member of the derived type (the "substruct"), casts between pointers of the two types, which are prohibited by the C standard in the general case, become legal.

As reported in the OWL 2 specification [85], OWL constructs can be grouped into families, which the library models as base types and then extends to concrete types as allowed by the pseudo-inheritance technique:

- **Axioms** (*CowlAxiom*): statements that specify what is true in the domain the ontology represents;
- **Class expressions** (*CowlClsExp*): sets of individuals identified by a formal specification of certain constraints on their properties;
- **Data ranges** (*CowlDataRange*): like class expressions, but for sets of *literals*, *i.e.*, data values such as strings or numbers;

- **Individuals** (*CowlIndividual*): instances of the knowledge domain, which may be *named* or *anonymous*;
- **Object property expressions** (*CowlObjPropExp*): relationships between pairs of individuals;
- **Data property expressions** (*CowlDataPropExp*): relationships between individuals and literals.

The data model has been carefully designed so that OWL constructs are *immutable*, *i.e.*, they cannot be modified after creation. Besides simplifying software design, immutability can be exploited to enable a whole family of performance optimizations that are critical for resource-constrained devices, as detailed in Section 2.5. The data model is built on top of *uLib*[4], an open source library that provides base data structures and utilities with an explicit focus on computational efficiency and low memory footprint.

Cowl uses the *reference counting* technique for memory management: every construct in the data model has a reference count attribute that is incremented when a new reference is created, and decremented when a reference is no longer needed, *e.g.*, when a referencing construct is destroyed. The idea is that a construct can be safely deallocated when its reference count reaches zero, as this entails that there are no more references pointing to it, and therefore it is no longer needed. Pseudo-inheritance easens the implementation of reference counting by allowing all constructs to share a common ancestor, *CowlObject*, which provides the necessary state and primitives.

Ontologies are modeled in Cowl through the *CowlOntology* object, which is essentially a collection of axioms. Under the hood, a *CowlOntology* is an optimized self-organizing in-memory store, which keeps axioms indexed by type and referenced entities, allowing for significant speed-ups when querying knowledge stored therein. The *CowlOntology* API provides facilities to add or remove axioms, annotations, and other constructs, allowing users to edit existing ontologies or create and populate new ones.

[4]uLib source code: `https://github.com/ivanobilenchi/ulib`

Ontologies can be queried through dedicated endpoints in the *CowlOntology* API, accepting *CowlIterator* instances as arguments. A *CowlIterator* is a C implementation of the *closure* pattern [63], a functional programming construct that allows bundling a function and its call site environment, enabling the function to reference local variables outside its scope. High-level languages supporting closures usually allow capturing the environment by simply referencing variables visible at the function call location. This is, however, not possible in C, therefore the programmer must explicitly pass the environment by providing a pointer to a generic *context* object. The *CowlIterator* function is called for each element matched by the query, and the presence of generic user-provided state allows for arbitrarily complex queries. Iteration can be stopped by returning `false`, enabling early termination for queries such as finding the first construct that matches some condition.

Ontologies can be read and written through the *CowlManager* object, which is able to handle many kinds of input sources and output targets. The whole I/O system is built on top of uLib streams, which unify access to files, memory buffers, network streams, and so on, as byte streams that can be read from and written to. Cowl's architecture allows for multiple readers and writers, either built-in or provided by the user. Readers (*CowlReader*) basically map from byte streams to sequences of OWL constructs, while writers (*CowlWriter*) can carry out the opposite transformation. The way OWL constructs are parsed and rendered depends on the specific ontology document format.

The base facility for ontology modularization in OWL 2 is the *imports* system, where ontologies can import other ontologies in order to gain access to their entities, expressions, and axioms. Imported ontologies may be retrieved from storage devices, network streams, or other input sources. Cowl delegates handling of imports to the user via the *CowlImportLoader* object, which must be implemented by returning the *CowlOntology* corresponding to a certain IRI. A similar approach is adopted for the management of parsing and serialization errors, where a user-specified *CowlErrorHandler* instance allows handling failures as fit for the application.

2.4 Axiom streams

One of Cowl's main novelty points concerns the way ontologies are read and written. Many existing OWL toolkits provide access to knowledge in an ontology document by deserializing it to an intermediate RDF data store. Likewise, serializing knowledge requires building the data store first, and then writing it as an OWL ontology document. This restriction stems from the close relationship between OWL and RDF, which allows OWL ontologies to be rendered as RDF graphs. Due to the inherent simplicity of the RDF language, each OWL axiom may need to be encoded by multiple RDF statements, possibly even scattered throughout the document, as there is no requirement on the adjacency of the triples that encode a specific axiom, as illustrated in Figure 2.2. In the example, three different axioms are serialized in *Functional* [85] and the RDF triple-based *Turtle* [90] syntaxes: in the former, byte (character) sequences encoding each axiom are consecutive, while they can be intermingled in the latter. This mandates the creation of an intermediate data store, which is populated and subsequently translated into a set of axioms when processing OWL ontologies serialized as RDF documents. Due to its simplicity and versatility for most applications, and likely to avoid fragmentation at the API level, this *store-based* approach to ontology processing has historically been adopted by OWL tools for both RDF-based and OWL-specific serializations. However, this technique proves to be wasteful for a number of scenarios: for example, if one already has a data model representation and wishes to render it as an OWL document, or is interested in a small subset of the axioms within an existing OWL document, avoiding the construction of a potentially large intermediate representation becomes desirable to save memory and minimize CPU usage.

The OWL functional syntax, as well as all standard OWL serialization formats, allows ontologies to be serialized in such a way that the resulting byte sequence respects the following structure:

- **Header:** containing IRI prefix declarations, the ontology IRI, import

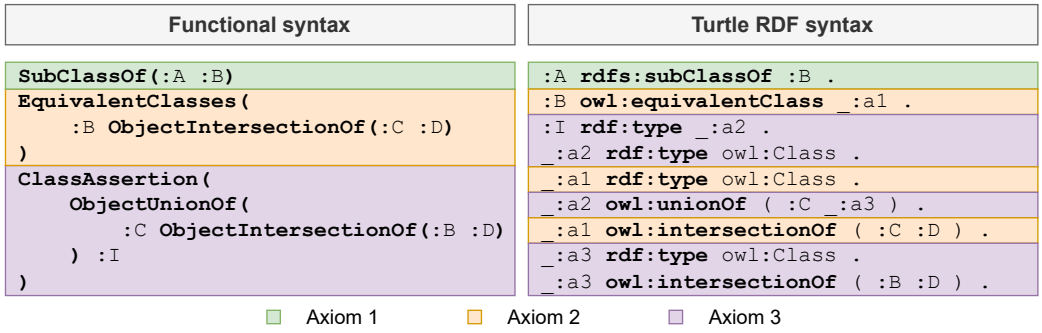


Figure 2.2: Valid encodings of three axioms in Functional syntax and Turtle. RDF provides no guarantees that triples belonging to each axiom are consecutive in the byte sequence.

statements, and ontology annotations.

- **Axioms:** a sequence of axiom definitions, usually the largest part of the byte sequence.
- **Footer:** a usually small and fixed sequence of bytes that closes the document.

Apart from the header and footer, it is clear that it is generally possible to serialize any data model as an ontology document by producing a suitable sequence of axioms, *i.e.*, an *axiom stream*. Cowl implements axiom stream serialization through the *CowlOStream* API, which allows user code to drive the serialization process by writing the ontology header, individual axioms, and finally the footer, without requiring the construction of an intermediate axiom store.

Even more interestingly, if the chosen OWL document format guarantees that the serialization of two different axioms can never be interleaved in the byte sequence, then ontologies can also be *read* as axiom streams. It is the case that OWL-specific serializations, including the functional, OWL/XML, and Manchester syntaxes, respect this constraint. Thus, the library implements this parsing technique through a separate API, *CowlIStream*: the user provides an arbitrary state pointer and a set of handler functions, which are called by the parser when the relevant construct is detected in the inbound byte stream.

Since the population of the *CowlOntology* data store is entirely bypassed, this approach to parsing is generally much more lightweight, as also evidenced by the experiments in Section 2.6. On the other hand, axiom stream parsing does not support ontology editing and programmatic queries, for which a *CowlOntology* store is needed, and it cannot be adopted in general for RDF-based serializations. However, it is worth noting that if the aforementioned property is enforced while serializing, *i.e.*, all triples that make up each axiom are written consecutively, then RDF documents may also be processed as axiom streams. This of course requires making the parser aware that the RDF document respects this property, via *e.g.*, an optional flag or a special comment at the beginning of the document. In any case, axiom stream parsing is not entirely ruled out for RDF, and may be efficiently implemented in future iterations of the library.

Combining axiom stream serialization and parsing with the byte stream capabilities of the *uLib* library proves to be especially effective in scenarios where devices must communicate semantically annotated information, such as in networked multi-agent systems. In such scenarios, the transmitter and the receiver can potentially exchange OWL knowledge graphs at the axiom level without ever materializing the entire data store on either end. An example of the interactions enabled by Cowl is showcased in Section 5.1.

2.5 Optimizations for embedded platforms

Most of the design and implementation choices made while developing the library are oriented towards an efficient use of hardware resources, with memory usage and code size being the primary focus, as they are the most heavily constrained aspects on embedded platforms. The techniques detailed hereafter are enabled by three core features exploited throughout the library, which have been described in Section 2.3: data model immutability, pseudo-inheritance, and reference counting.

- **Instance sharing:** objects that are likely to be referenced by multiple

axioms are stored in an instance pool, and constructor functions are designed to return shared instances from this pool. This technique is also conditionally exploited for strings representing common language constructs, ensuring that duplicate instances of such strings are never created.

- **Optimized IRI storage:** IRIs can be split into a namespace and a local name. Since namespaces are often shared among multiple IRIs, Cowl internally represents IRIs with two strings: the namespace and the remainder. Every time an IRI is detected in the byte stream, Cowl extracts the namespace according to the *XML Namespaces* specification [23] and adds it to the string instance pool. This allows multiple IRIs to share references to the same namespace instance, significantly reducing the memory required to store them. The technique is sketched in Figure 2.3.

- **Faster equality checks:** instance sharing and immutability are exploited to speed up equality checks: by definition of instance sharing, equal objects must be the same instance, therefore equality checks can be performed as pointer comparisons, eliminating the need for more costly comparisons of object contents. Speeding up equality checks is crucial, as the library makes extensive use of hash-based data structures, where they are needed for all CRUD (Create, Read, Update, Delete) operations. A further optimization is implemented for strings, whose hashes are cached and used to expedite checks: a proper hash function[5] guarantees that identical strings have equal hashes, therefore comparisons can be avoided for strings with different hash values, eliminating the need for the vast majority of them.

- **Code size reduction:** even though Cowl's data model reflects the whole OWL 2 structural specification type-wise, most types and their related APIs are placeholders or thin wrappers for a shared low-level implementation. Since most OWL constructs can be viewed as structurally equivalent to typed tuples of other constructs, Cowl implements

[5]Cowl uses the DJBX33A [36] hash function for strings.

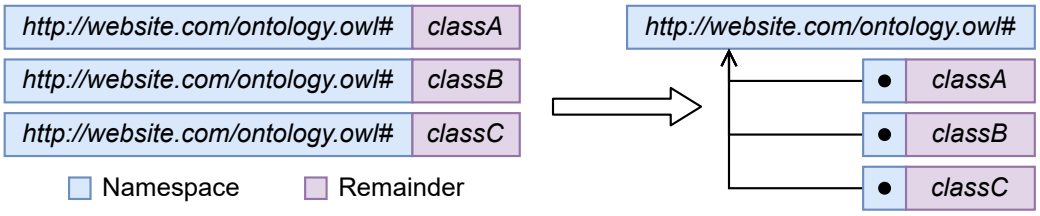


Figure 2.3: Optimized IRI storage: namespace instances are shared among IRIs.

them as a single *CowlComposite* structure holding a variable number of *CowlObject* fields, as shown in Figure 2.1. All types that inherit from *CowlComposite* share the same implementation for facilities such as field getters, equality checks, hash computation, field enumeration, and so on. This greatly reduces code size without sacrificing type safety at the API level.

- **Low-level optimizations:** further memory savings are allowed by a number of optimizations on low-level types. The size of numeric types can be controlled at compile time, so that integers can vary between two, four, and eight bytes, and floats between four and eight bytes, with smaller representations being obviously desirable for resource-constrained devices. This of course introduces compromises for certain implementation details, such as the maximum size of collections, or the maximum number of references that can be held for any object. Strings and vectors make use of the *small buffer optimization*, a technique where buffers that are smaller than the size of the enclosing structure are stored directly within it, without allocating additional memory on the heap. Cowl also implements the *pseudo-generics* technique for collections: type definitions and code for collections of different data types can be generated by means of a set of preprocessor directives, mimicking a template system. This introduces a trade-off between runtime performance, in terms of both speed and memory overhead, and code size, therefore specialized versions of collections are only generated for frequently used and performance-critical data structures.

2.6 Evaluation

Performance evaluation is necessary to validate Cowl's suitability for the heterogeneous contexts and platforms required by the SWoE. A preliminary experimental campaign was carried out in [14], comparing an alpha version of the library to state-of-the-art OWL ontology manipulation toolkits. Results of the evaluation are recalled in Section 2.6.1, before reporting those of a further evaluation of a stable version of the library on a popular microcontroller platform, as detailed in Section 2.6.2.

For both campaigns, the evOWLuator framework (see Chapter 4) was leveraged for test automation, exploiting its ability to invoke command line tools over arbitrary OWL datasets and gather performance metrics concerning turnaround time and memory usage.

2.6.1 Laptop tests

An alpha version of the library has been compared to the following state-of-the-art OWL tools: OWL API (version 5.1.20), Owlready2 (0.37), and owlcpp (0.3.3). To be integrated into evOWLuator, all libraries have been wrapped in straightforward command line tools that accept the path to the ontology as an argument. The testbed is a 2021 Apple MacBook Pro 16"[6]. All reported performance results are the average of 5 cold runs.

The dataset used for the tests consists of 109 knowledge bases, extracted from the 2014 *OWL Reasoner Evaluation Workshop* (ORE2014) competition corpus[7] as follows: all ontologies in functional-style have been considered and sorted by size, then one sample has been extracted at each MiB boundary (if available). Tests have concerned ontology parsing and querying time, and peak memory usage. The following queries have been selected for benchmarking: **(Q1)** retrieval of all axioms in the ontology; **(Q2)** retrieval of all classes in

[6]Apple M1 Max System-on-Chip with 64 GB RAM, 1 TB SSD, macOS Monterey 12.3.
[7]ORE2014 corpus: `http://dl.kr.org/ore2014`

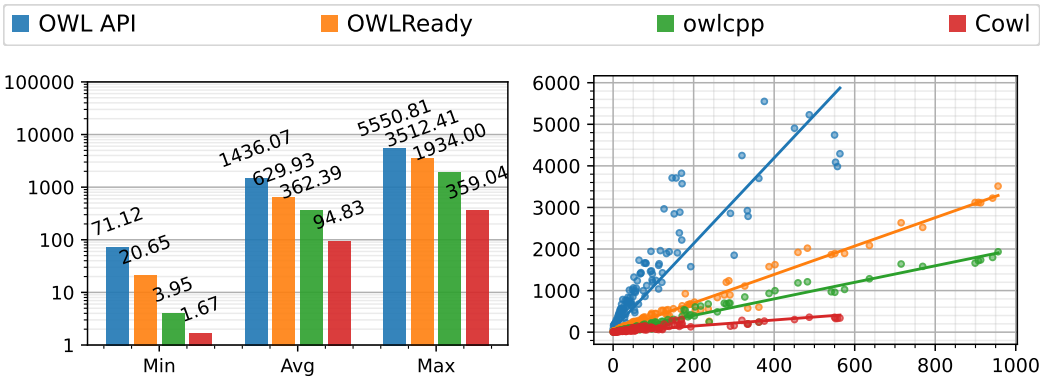
the ontology and, for each class, retrieval of all subclass axioms directly referencing it as the subclass or superclass.

Before presenting the results of the experimental campaign, it is important to point out a few key differences among the evaluated tools:

- They support different OWL serializations: the OWL API features parsers for most syntaxes in literature; Owlready2 only supports RDF/XML, OWL/XML and Turtle; owlcpp is limited to RDF/XML, while Cowl currently only features a functional syntax parser. The libraries have therefore been configured as follows: Owlready2 and owlcpp have been run on RDF/XML documents, while the OWL API and Cowl have processed their functional syntax variants. This is apparent from data points in the scatterplots, spanning larger ontology sizes for tools using the RDF/XML serialization.
- They adopt different data model architectures: the OWL API and Cowl provide a direct mapping to OWL axioms and constructs, while Owlready2 and owlcpp store ontologies in RDF triple stores. This entails that ontology queries have considerably different implementations, with significant influence on performance.

Figure 2.4 shows peak memory usage metrics, computed by evOWLuator as the *maximum resident set size* (MRSS) of the process. Cowl exhibits a significantly lower memory footprint than all other tools, with the margin being especially wide when compared to the non-native OWL API and Owlready2 libraries. All libraries follow a roughly linear trend for memory occupancy in relation to ontology size, even though absolute values on the most resource-constrained nodes –such as nodes classified as Class 0 and Class 1 by the Internet Engineering Task Force (IETF) [22]– may be different from the 64 bit testbed workstation.

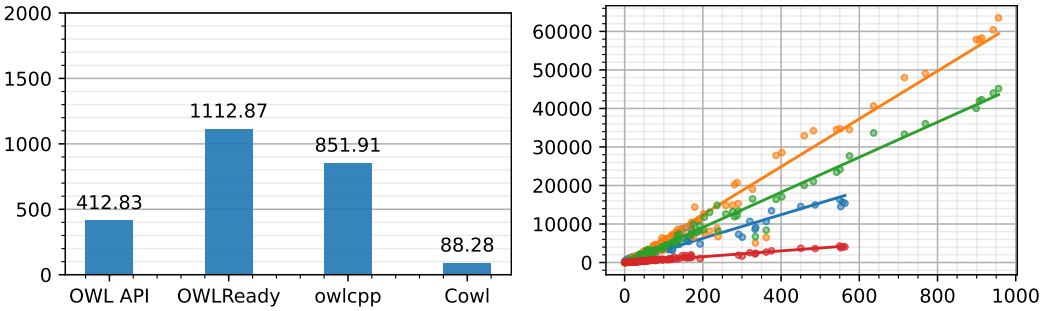
Figure 2.5 reports results on turnaround times for ontology parsing. Similar considerations apply here, with Cowl outperforming the other tools, and all libraries behaving linearly in relation to ontology size. The OWL API performs



(a) Dataset-wide min, avg and max (*MiB*).

(b) Memory peak by ontology size (*MiB*).

Figure 2.4: Comparison of peak memory usage.



(a) Cumulative time (*s*).

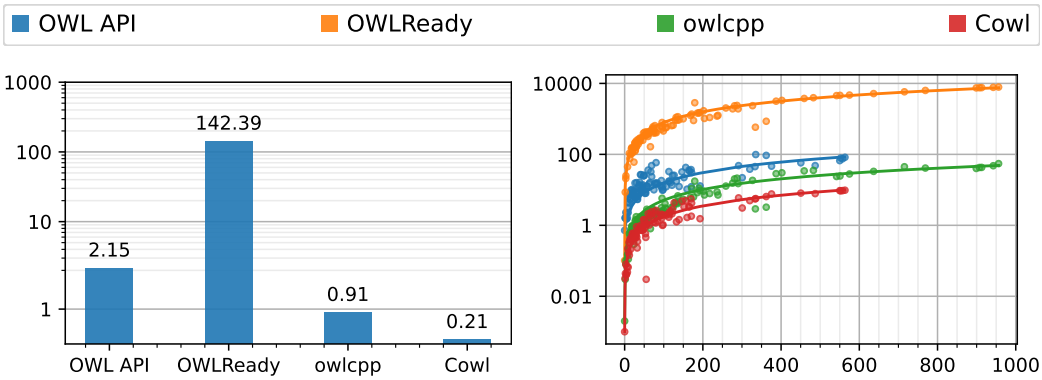
(b) Time (*ms*) by ontology size (*MiB*).

Figure 2.5: Comparison of parsing time.

remarkably well with respect to parsing time, even outperforming the native owlcpp library, albeit at the cost of a consistently and significantly higher memory usage than all other tested toolkits.

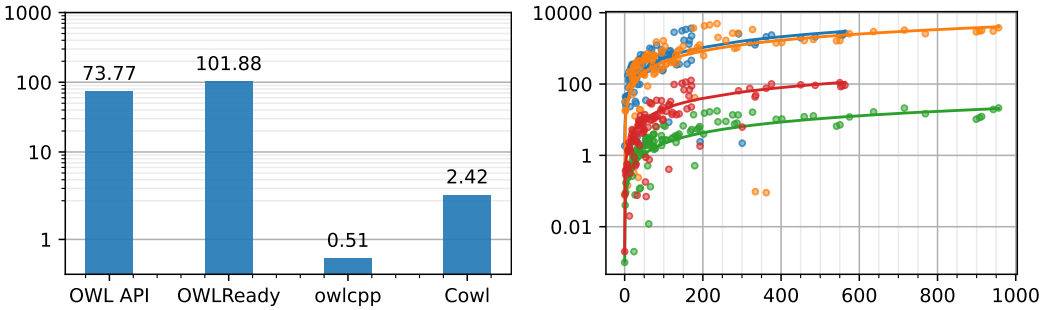
Figure 2.6 displays turnaround times for the retrieval of all axioms. Cowl is consistently faster than the other frameworks; interestingly, the OWL API is almost able to match owlcpp. As the query entails iterating over all constructs in the KB, triple store based tools are penalized, as they must process a larger number of constructs, since OWL axioms are generally encoded through multiple RDF statements.

Figure 2.7 shows turnaround times for the retrieval of all subclass axioms for all classes: owlcpp is the top performer, followed by Cowl, while the OWL API and Owlready2 exhibit very similar, much higher figures. This



(a) Cumulative time (s). **(b)** Time (ms) by ontology size (MiB).

Figure 2.6: Comparison of time taken to retrieve all axioms.



(a) Cumulative time (s). **(b)** Time (ms) by ontology size (MiB).

Figure 2.7: Comparison of time taken to retrieve all subclass axioms for all classes.

query was deliberately chosen to be easily answered by triple stores (through *e.g.*, the SPARQL query in Listing 1), as it strongly relies on access to constructs that directly reference specific entities. Axiom-based tools, on the other hand, must populate auxiliary indices to speed up references to specific OWL entities, and may still end up processing unrelated axioms, depending on the indexing strategy. Performance of such queries can be improved by introducing additional indices or more sophisticated data structures, though this usually comes at the expense of memory usage, therefore it was not deemed a worthwhile trade-off due to Cowl's focus on memory-constrained platforms.

```
SELECT *
WHERE {
    { ?a_class rdfs:subClassOf ?b_class . }
    UNION
    { ?b_class rdfs:subClassOf ?a_class . }
    FILTER (!isBlank(?a_class))
}
```

Listing 1: Retrieval of all subclass axioms for each class from a triple store.

2.6.2 Embedded board tests

While the previous results highlighted state-of-the-art performance with respect to other popular OWL toolkits, those tests were carried out on a laptop testbed. Proving that the proposed system can really permeate the SWoE device spectrum requires evaluations on a suitably small nano-scale device. Furthermore, comparing the novel *axiom-based* approach to ontology manipulation with the traditional *store-based* technique is crucial to really understand its usefulness in SWoE contexts.

Arduino[8] boards are among the most popular microcontroller platforms. Their open-source nature, combined with a vast community and availability of compatible hardware and software libraries, make them a versatile environment for all kinds of applications, spanning from simple automation projects to complex robotics and IoT infrastructures. As such, the Arduino Due board[9] has been chosen as the testbed for a comprehensive experimental campaign. To the best of our knowledge, this is the first time a fully capable OWL library is deployed and evaluated on a microcontroller this small.

The experimental setup is illustrated in Figure 2.8. The board has been connected to a PC, and the evOWLuator framework has been used to manage all experiments. The framework's modular architecture and support for running reasoning tasks on remote devices (see Section 4.4) have been exploited by implementing a plugin that lets the PC communicate with

[8]Arduino home: `https://arduino.cc`

[9]Equipped with 32 bit ARM Cortex M3 CPU clocked at 84 MHz, 96 KiB of SRAM, and 512 KiB of flash storage.

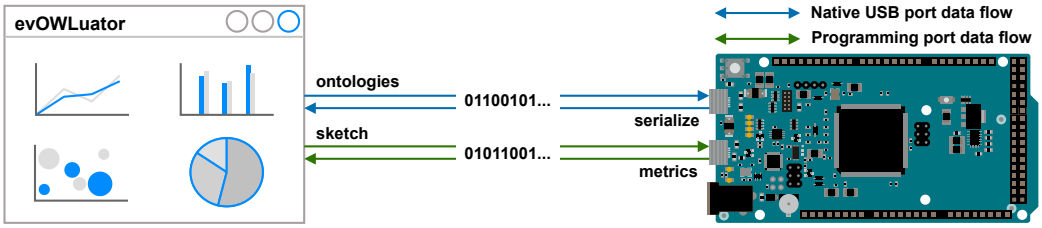


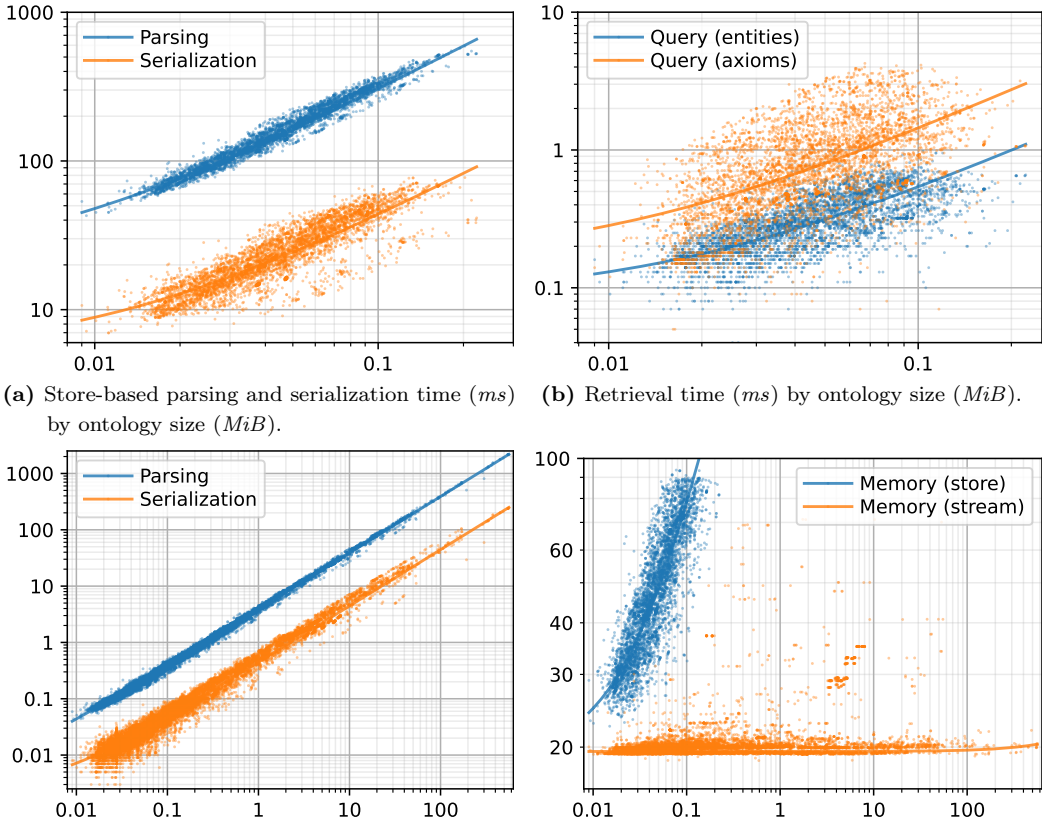
Figure 2.8: Experimental setup for tests on Arduino Due.

the board using both its UART[10] and USB[11] 2.0 ports. More precisely, the UART programming port has been used to flash Arduino "sketches" responsible for the execution of on-board tests involving the Cowl library, and to report performance metrics back to the evaluation framework. Concurrently, the faster USB port has been leveraged for larger data transfers, such as loading serialized ontology documents to and from the board. Since Cowl's ontology I/O primitives are built on top of *uLib* streams, reading and writing ontologies through the USB port has just required the implementation of custom input/output streams that leverage the built-in Arduino *USBSerial* class.

The dataset used for the experiments is the whole 2014 OWL Reasoner Evaluation Workshop (ORE2014) [6] competition corpus, consisting of 16555 ontologies in functional syntax, ranging from 10 KiB to over 500 MiB in size. Tests focus on ontology parsing, serialization, and querying time, and peak memory usage. For parsing and serialization times, the time required to transfer the ontology document over the USB port is factored out in the results reported hereafter, since it strictly depends on the port's transfer speed and it dominates the overall time, making it hard to understand how much time is actually spent processing the ontology.

The first test has concerned loading ontologies from the USB port into the in-memory data store, querying them, and serializing them back to the USB port. Figure 2.9a displays, on a logarithmic scale, the time required to parse and serialize ontologies as a function of their size. Both parsing and

[10]Universal Asynchronous Receiver-Transmitter.
[11]Universal Serial Bus.



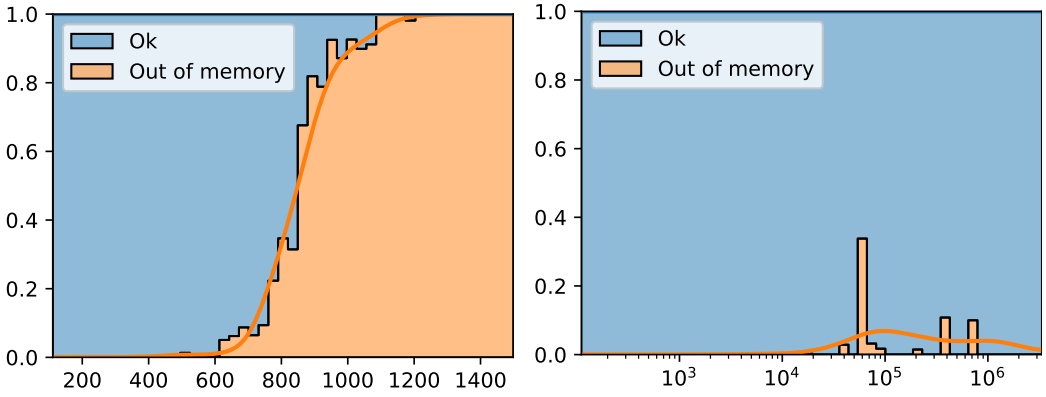
(a) Store-based parsing and serialization time (ms) by ontology size (MiB).

(b) Retrieval time (ms) by ontology size (MiB).

(c) Stream-based parsing and serialization time (s) by ontology size (MiB).

(d) Store- vs stream-based memory usage (KiB) by ontology size (MiB).

Figure 2.9: Performance metrics on Arduino Due.



(a) Store-based approach.

(b) Stream-based approach.

Figure 2.10: Ratio of out-of-memory events by axiom count in ontology parsing.

serialization times show a linear trend, as expected. The board has been able to process 9543 ontologies (57.6% of the whole dataset), with the largest ontology being 228 KiB. The remaining ontologies could not be processed due to memory exhaustion.

Figure 2.9b plots the time spent to retrieve: (i) all the entities within an ontology; (ii) for each ontology class, all subclass axioms directly referencing it as the subclass or superclass. The first query can be carried out in less than 1 ms even on the largest ontologies, while the second, more complex query takes about 3 ms at most. This result demonstrates that the architecture of the *CowlOntology* store, combined with the optimizations described in Section 2.5, allows efficient queries even on devices with strict processing limitations.

The second test has involved processing the ontology document as an axiom stream, while simultaneously serializing detected axioms. In this specific case, since the USB 2.0 port is not capable of full-duplex communication, serialization has been carried out over a dummy byte stream which discards its output. In this configuration, Cowl is able to process nearly the entire ontology dataset (16464 ontologies, 99.5%), as shown in Figure 2.9c: the largest processed ontology is also the largest ontology in the dataset, with a size of 563 MiB. This is mainly due to fact that ontologies are processed one axiom at a time, removing the need for the intermediate data store, which results in a substantial reduction in memory usage. This advantage becomes even more apparent when comparing peak memory usage, depicted in Figure 2.9d. While the conventional approach is limited by the size of the ontology being processed, the boundary of the stream-based strategy is solely determined by the size of the largest axiom in the ontology. As such, the memory usage trend changes from linear to roughly constant, with only a few exceptions, thus creating the potential to manage ontologies of much greater size.

Another noteworthy observation emerges when comparing the rate of out-of-memory errors between the store- and stream-based approaches. Figure 2.10 is the histogram plot of ontologies grouped by axiom count (x-axis) over the

percentage of ontologies that result in an out-of-memory (OOM) event within each bin (y-axis). As illustrated in Figure 2.10a, in the store-based approach the ratio of OOM events steadily increases towards the 1.0 limit for ontologies with more than ∼1100 axioms. This behavior is consistent with the limited availability of SRAM memory, which quickly gets saturated by the in-memory store. Conversely, as shown in Figure 2.10b, the stream-based approach enables the handling of ontologies with a very large number of axioms, greatly minimizing the chances of experiencing memory saturation.

This experimental campaign has demonstrated the capability of the Cowl library to handle OWL knowledge graphs on embedded devices, indicating that it can efficiently manage and process ontologies, even under the constrained resources typical of embedded systems. Its architecture and optimizations enable it to handle a substantial portion of the dataset through the classical store-based approach, allowing embedded applications to query and edit moderately sized ontologies. The stream-based technique, other than being useful on its own for a subset of use cases, can act as a practical fallback when memory limitations do not allow keeping larger ontologies entirely in memory, allowing the tool to scale up to graphs of virtually any size. These results collectively suggest that Cowl's architecture, optimizations, and novel processing techniques effectively address the challenges of OWL ontology management on limited-capacity devices, broadening the pervasivity of SWoE applications.

Chapter 3

Tiny-ME: a reasoning engine for the Semantic Web of Everything

This chapter introduces *Tiny-ME*[1] [94] (the Tiny Matchmaking Engine), a SWoE-oriented matchmaking and reasoning engine. It features a multiplatform architecture designed from the ground up, with a common core in C language granting both portability and efficient implementation of KB management primitives and reasoning services. On top of the low-level core, multiple high-level APIs are provided to facilitate integration in a variety of platforms and technological stacks, ranging from the Web to tiny resource-constrained devices.

The reasoner provides standard (Ontology Classification, Ontology Coherence, Concept Subsumption, and Concept Satisfiability) and non-standard (Concept Abduction, Contraction, Bonus, Difference, and Covering) inference services in an OWL 2 subset corresponding to the $\mathcal{ALN}(\mathcal{D})$ Description Logic (*Attributive Language with unqualified Number restrictions and Datatypes*). All major desktop and mobile operating systems are compatible out of the box, as well as all major browser platforms through a port of the system to the *WebAssembly*[2] runtime, [91] allowing its use in client-side Web applications. Tiny-ME can also be used for reasoning in microservice architectures for Web and cloud applications: to this aim, the standard *OWLlink* [69] protocol

[1]Tiny-ME: `http://swot.sisinflab.poliba.it/tinyme/`
[2]WebAssembly: `https://webassembly.org/`

for remote reasoner invocation has been extended, adding support for the provided non-standard inference services. Architectural and technological design choices allow the system to be also deployed on nano-scale devices, such as embedded drone autopilots and microcontrollers. Such platform diversity is a first in the OWL reasoning landscape, making the system a candidate for the development of a practical SWoE infrastructure.

The remainder of this chapter is as follows: after background in Section 3.1, Section 3.2 discusses the supported inference services; Section 3.3 describes the main principles that drove the design of the system and details its architecture and main data structures; Section 3.4 outlines platform-specific and Web APIs that allow easier cross-platform integration; Section 3.5 details updates to the system extending expressiveness beyond $\mathcal{ALN}$; finally, Section 3.6 presents the results of experiments on desktop, mobile, browser, and microcontroller platforms.

3.1 Background

Classical Semantic Web contexts are characterized by the consistent availability of substantial computational and networking resources. In contrast, the SWoE operates under markedly different conditions, with resource-constrained hardware and micro-devices dispersed across various physical locations, acting as sources of information. The availability of these devices is highly variable and unpredictable, influenced by factors such as user and device mobility, constraints of wireless communication links, and energy supply limitations. Consequently, executing batch workloads on highly expressive and complex Knowledge Bases is out of place in SWoE contexts. Instead, inference engines must be capable of rapidly processing queries on smaller and less complex annotations. This shift requires a reconsideration of the expressiveness of the adopted logical languages, tailoring them to suit these specific operational constraints. Historically, the intractable worst-case complexity of OWL DL has led to the development of OWL 2 *profiles*. These profiles simplify the language and the admissible KB axioms, enabling the use of efficient algo-

rithms while maintaining sufficient expressiveness for practical applications. $\mathcal{EL}^{++}$ [3] expanded upon the basic $\mathcal{EL}$ (*Existential Language*) Description Logic to accommodate a range of applications, particularly those involving large ontologies with moderate expressiveness. The proposal introduced a polynomial-complexity Ontology Classification algorithm, fostering the creation of high-performance $\mathcal{EL}^{++}$ classifiers such as *ELK* [57]. Similarly, one of the initial methods for adapting non-standard inferences like Concept Contraction and Concept Abduction to pervasive computing [92] involved implementing structural algorithms on acyclic TBoxes within the $\mathcal{AL}$ (Attributive Language) DL through a mobile Relational Database Management System (RDBMS) query layer.

The initial wave of mobile and IoT-focused inference engines adopted streamlined methods compared to traditional Semantic Web reasoners to cope with the limited memory capacity of devices. *Pocket KRHyper* [112], 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* [2] reasoner implemented a simple resolution and pattern matching algorithm on a subset of OWL-Lite. Analogously, *MiRE4OWL* [58] 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 [41] consequence-driven $\mathcal{EL}^{+}$ reasoning was ported to a Programmable Logic Controller (PLC) for industrial automation. The modular rule-based reasoner in [118] combined selective rule loading and a two-stage RETE algorithm, exhibiting satisfactory performance on the *Sun SPOT* sensor platform for small- and medium-sized ontologies.

A second category of reasoners includes those initially developed for traditional computer platforms and later adapted for mobile devices, capitalizing on the growing availability of computational resources in smartphones and tablets. As detailed in [18], five Java-based OWL reasoners were modified for use on Android, though this adaptation required significant effort. In

[56], the ELK reasoner was re-engineered to transition from Java Standard Edition (SE) to Android, minimizing memory usage and adding support for multi-core CPU architectures. Beyond adaptations, native mobile inference engines have also been developed. For instance, *Mini-ME 2.0* [107] is designed specifically for Android, functioning as both a *matchmaker* and a reasoner, and maintains compatibility with Java SE. In [126] a rule engine for Android adopts the novel $RETE_{pool}$ algorithm on OWL 2 RL rulesets, capable of balancing memory usage and time performance. *Mini-ME Swift* [98] is the first OWL reasoner for iOS, re-designed from the above Mini-ME using the OWL API for iOS [96], an iOS-specific knowledge manipulation library.

The World Wide Web iself is a prevalent deployment environment for new software tools and applications, and the browser has become the default channel for many relevant scenarios [119]. Web applications typically exploit remote services for DL reasoning. The Web-oriented JavaScript (JS) language has been seldom chosen for reasoning engine development, primarily due to its inferior performance compared to other programming languages, and because it has been deemed simpler to invoke DL inferences through client-server protocols such as OWLlink [69], which is supported by many Semantic Web reasoners. A large number of semantic-enabled Web applications for conventional desktop clients follow this approach, covering a wide range of domains; unfortunately, both technological and user interface design choices prevent their adaptation to mobile and ubiquitous contexts. Notable semantic-enabled mobile-oriented Web apps include the *DBpedia Mobile* [8] and *LOD4AR* [132] location-based discovery clients for mobile devices. *OntoWiki Mobile* [35] is a mobile knowledge management applications exploiting HTML5 and jQuery Mobile to locally store knowledge annotated by users in the field, even without Internet connection, and synchronize it with a remote server when a connection becomes available. Overall, the survey in [138] on semantic-enabled mobile apps has found that only 4 out of 36 apps had been developed with Web-based cross-platform technologies.

Efforts on using client-side Web technologies for cross-platform reasoning engines are sparse. The *EYE* (Euler Yet another proof Engine) [131] is a

notable example, capable of forward and backward rule chaining over Euler paths. It has been adapted to JS-based environments such as the Web and *Node.js*,[3] as made possible by employing a JS porting of the *SWI Prolog* [136] engine. The *MobiBench* mobile semantic rule engine evaluation framework [127] facilitates the integration of rule engines developed in JS for the Web and other platforms. This integration is achieved through the *Apache Cordova*[4] Software Development Kit (SDK). *HyLAR+*[5] [121] is a hybrid OWL 2 EL reasoner: both the server side and the client side can execute reasoning tasks by running the same JS code with Node.js and *AngularJS*[6], respectively. It leverages the *JSW* semantic technology framework for JS[7], which includes the *BrandT* browser-hosted OWL 2 EL inference engine [116].

WebAssembly [91] discloses a viable path to developing or porting reasoning engines to Web applications and stand-alone runtime environments like Node.js with minimal performance penalty. Proposals, however, are in very early stages. *EYE JS*[8] is an initial WebAssembly port of EYE.

Table 3.1 summarizes relevant features of related reasoning systems: with respect to semantic matchmaking functionality as recalled in Section 1.2.2, the "*Full*" label refers to systems supporting only *exact* or full/subsume match degrees [66], 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]Node.js: `https://nodejs.org/`
[4]Apache Cordova home: `https://cordova.apache.org/`
[5]HyLAR GitHub repository: `https://github.com/ucbl/HyLAR-Reasoner`
[6]AngularJS home: `https://angularjs.org/`
[7]JSW GitHub repository: `https://github.com/JS-WindowFramework/JSW`
[8]EYE JS GitHub repository: `https://github.com/eyereasoner/eye-js`

Table 3.1: Features of related reasoning systems

	Features				Platforms				
Name	**DL**	**Algorithm family**	**Matchmaking**	**Explanation**	**Language**	**Web/Cloud**	**Desktop**	**Mobile**	**Embedded**
μOR [2]	OWL-Lite$^-$	Pattern matching	Full		Java		JVM		JamVM
COROR [118]	$\mathcal{SHOIN}(\mathcal{D})$	RETE	Full		Java		JVM		Sun SPOT
ELK [56]	$\mathcal{EL}^+$	Consequence-based			Java		JVM	Android	
HermiT [38, 18]	$\mathcal{SROIQ}(\mathcal{D})$	Tableaux	Full		Java	OWLLink	JVM	Android	
JFact [18]	$\mathcal{SROIQ}(\mathcal{D})$	Tableaux	Full		Java	OWLLink	JVM	Android	
Konclude [115]	$\mathcal{SROIQV}(\mathcal{D})$	Hybrid	Full		C++	OWLLink	Native		
Mini-ME [107]	$\mathcal{ALN}$	Structural	Full, Approx.	Yes	Java	OWLLink	JVM	Android	
Mini-ME Swift [98]	$\mathcal{ALN}$	Structural	Full, Approx.	Yes	Swift		macOS	iOS	
MiRE4OWL [58]	$\mathcal{SHOIN}(\mathcal{D})$	RETE	Full		C++		Native	Windows	
Pellet [18]	$\mathcal{SROIQ}(\mathcal{D})$	Hybrid	Full	Yes	Java		JVM	Android	
PLC-based [41]	$\mathcal{EL}^+$	Consequence-based			SCL				SIMATIC
Pocket KRHyper [112]	$\mathcal{ALC}^+$	Tableaux	Full		Java		JVM	J2ME	
RETEpool [126]	OWL-RL	RETE	Full		Java		JVM	Android	
Tiny-ME	$\mathcal{ALN}(\mathcal{D})$	Structural	Full, Approx.	Yes	C, Obj-C, Java	OWLLink, JavaScript	Native, JVM	Android, iOS	Any with C support

3.2 Inference services

The first release of Tiny-ME [94] has targeted the $\mathcal{ALN}$ DL, which has been recalled and motivated in Section 1.1.1. It implements polynomial-complexity structural algorithms on $\mathcal{ALN}$ concept expressions. This section concerns the algorithmic implementation of the inference services, which have been formally introduced in Chapter 1.

As recalled in Section 1.1.2, in the $\mathcal{ALN}$ DL, comparisons between unfolded and normalized concept expressions basically come down to set operations. As an example, let us consider the following $\mathcal{ALN}$ TBox:

$$A \equiv \forall P.D \sqcap \geq 3P$$
$$B \sqsubseteq \forall P.D$$
$$C \sqsubseteq B \sqcap \geq 2P$$
$$E \sqsubseteq D$$
$$F \equiv E \sqcap \neg D$$

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

$$A \rightarrow \forall P.D \sqcap \geq 3P$$
$$B \rightarrow B \sqcap \forall P.D$$
$$C \rightarrow B \sqcap C \sqcap \forall P.D \sqcap \geq 2P$$
$$D \rightarrow D$$
$$E \rightarrow E \sqcap D$$
$$F \rightarrow \bot$$

It can be noticed B and C appear among the conjuncts of their own unfolded concept expressions, while A does not: as explained in [4, §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.

Another noteworthy consideration concerns F, whose concept expression became $\bot$: F was originally defined as the intersection of E and $\neg D$, which unfolds to $E \sqcap D \sqcap \neg D$, resulting in a clear contradiction. Its concept expression is therefore collapsed to $\bot$ while computing the CNF, entailing that the F concept is not **satisfiable**.

Finally, let us suppose we need to check whether **subsumption** holds between A and B: 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 $(\geq 2P)$. The exact algorithm for subsumption will be described in what follows.

Preprocessing

Unfolding and **CNF normalization** are crucial preprocessing steps, therefore care has been devoted to their optimization. In [98], both were improved by caching completely unfolded and normalized concepts. However, while designing Tiny-ME it quickly became clear that caching could be extended to *intermediate unfolded concepts* as well [94]: 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.

Satisfiability and Subsumption

Satisfiability is trivially checked by performing CNF normalization. As recalled in Section 1.1.2, a CNF-normalized concept expression A is either $\bot$, or the conjunction 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 [4]. 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, $R \sqsubseteq S$.

Concept Contraction

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 $x > 0$ 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 example TBox:
$\langle G, K \rangle = CC(A, \leq 2\ P) = \langle \geq 3\ P, \geq 2\ P \rangle$.

Concept Abduction and Bonus

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 $\not\exists\ B$ 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 example TBox: $H = CA(A, B) = \geq 3\ P$.

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.

Concept Difference

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 the example TBox:
$CD(B, A) = B$ and $CD(A, B) = \geq 3\ P$.

Concept Covering

The structural algorithm for CCov, given a set $\{R, S_1, S_2, ..., S_n\}$ of unfolded and CNF-normalized concept expressions, 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.

Ontology Classification

Tiny-ME adopts a variant of the *enhanced traversal* algorithm in [5], which also accounts for subsumption cycles detected while preprocessing the ontology. Subsumption check results are *cached*, as customary for OWL reasoners, though significant effort was devoted in avoiding checks whenever possible: classification is performed according to the concept *definition order* [5], which allows skipping the costly *bottom search* step of the traversal algorithm for primitive concepts having acyclic descriptions. The exploitation of *told disjoints* [124] and *told subsumers* has been implemented by pre-populating the subsumption cache according to the explicitly stated subclass, class equivalence, and class disjointness axioms. Moreover, the implementation of *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 [107]. However, previous experiments [98] have demonstrated that this method can be time-intensive, especially for larger TBoxes. One possible solution to mitigate this issue is the implementation of aggressive caching policies for unfolded concepts, although this comes at the cost of increased memory usage. Tiny-ME addresses this challenge by executing the Coherence check using a modified version of the Classification algorithm, which halts immediately upon detecting an unsatisfiable concept. Since normalization in Tiny-ME is *lazy*, *i.e.*, it occurs only when necessary for an inference task, and considering that the Classification algorithm actively avoids costly subsumption checks as much as possible, the total number of required normalizations is also diminished. Consequently, this approach

typically yields a substantial enhancement in both time and memory efficiency compared to the naive method. Furthermore, the Coherence check of a TBox $\mathcal{T}$ can be skipped in the following cases:

- $\mathcal{T}$ is *trivially incoherent* if $\exists$ a concept expression C in $\mathcal{T} \mid C \sqsubseteq \bot$;
- $\mathcal{T}$ 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.

3.3 Architecture

In designing a reasoner capable of adapting to a broad spectrum of use cases, several challenges emerge: the system must be flexible enough to cater to diverse needs, user-friendly and straightforward to maintain, while also being highly resource-efficient. Overlooking any one of these critical requirements significantly hampers its successful application in SWoE scenarios. In order to meet the above goals, the design of Tiny-ME has followed a few fundamental criteria:

- Inference services ought to be implemented only once and designed to function across the broadest possible range of platforms. This approach is vital because porting reasoning algorithms, especially when optimization comes into play, poses significant challenges. By adopting shared implementations, the maintainability of the overall system can be substantially enhanced, streamlining development while ensuring consistency and efficiency of algorithm outcomes and performance across different environments.

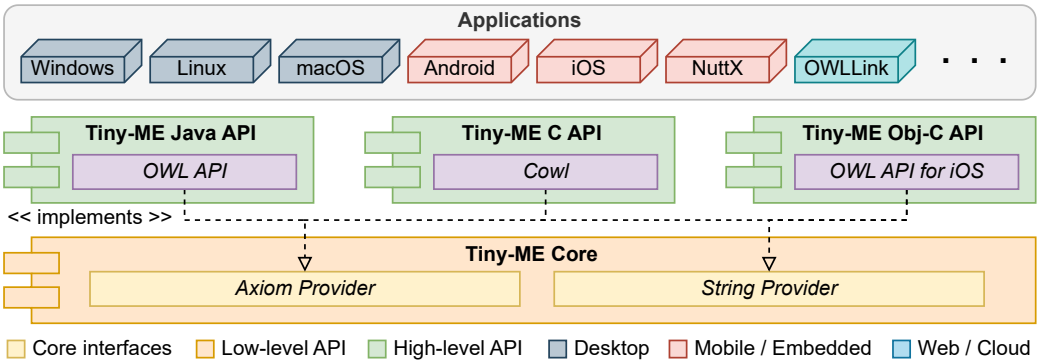


Figure 3.1: Tiny-ME high-level architecture.

- The implementation of inference services must be very efficient, particularly regarding memory usage. This is crucial for enabling deployment on small, resource-constrained devices, where optimal utilization of limited memory is essential to effectively support useful inferences.

- Applications should be provided simple, language-specific APIs to interact with the reasoner, and it must be feasible to develop one for a new language with relative ease, leveraging the existing implementation of inference algorithms. This approach guarantees that the system maintains both flexibility and user-friendliness, accommodating a wide range of development needs and preferences.

- It is advisable to segregate knowledge representation (KR) and reasoning functionalities into distinct modules. KR incurs substantial computational resource costs, as OWL data models and parsers are typically extensive in terms of both code and data. In prevalent interchange syntaxes for the Semantic Web, language constructs are essentially strings, which take up significant amounts of memory and are sometimes not strictly necessary for reasoning procedures. Additionally, the OWL 2 specification mandates support for the RDF/XML [103] serialization, requiring the inclusion of XML parsers and further increasing code size and memory demand.

High-level architecture

The overall high-level architecture of Tiny-ME is reported in Figure 3.1 and described as in what follows:

- **Core layer:** its implementation adheres to standard C11 without relying on compiler extensions or platform-specific API calls. The layer provides highly optimized standard and non-standard inference algorithms, supported by their respective data structures, and it is crafted to remain independent of the way in which knowledge is represented or stored. $\mathcal{ALN}$ OWL entities (i.e., named constructs like classes, object properties, and named individuals) are represented as numerical identifiers, called *entity pointers*, and their string representation is never required during the reasoning process. The conversion from structured KB axioms to expressions of entity pointers is carried out through the *Axiom provider* interface, which the reasoner consults when populating its data structures. An optional *writer* API is able to carry out the inverse transformation, by interacting with the *String provider* interface for acquiring the string representations of entity pointers. In a distributed SWoE architecture, some devices may lack the need or memory capacity for string representation capabilities. In such scenarios, they can solely implement the Axiom provider API and still effectively perform reasoning tasks on (unlabeled) entities.
- **Platform-specific APIs:** the modularity of the architecture facilitates the implementation of multiple APIs across various programming languages, as detailed in Section 3.4. Generally, the adoption of C11 for the reasoning core paves the way for an array of potential higher-level APIs. This is primarily because the runtimes of most programming languages provide, at a minimum, 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 is available. It is notewor-

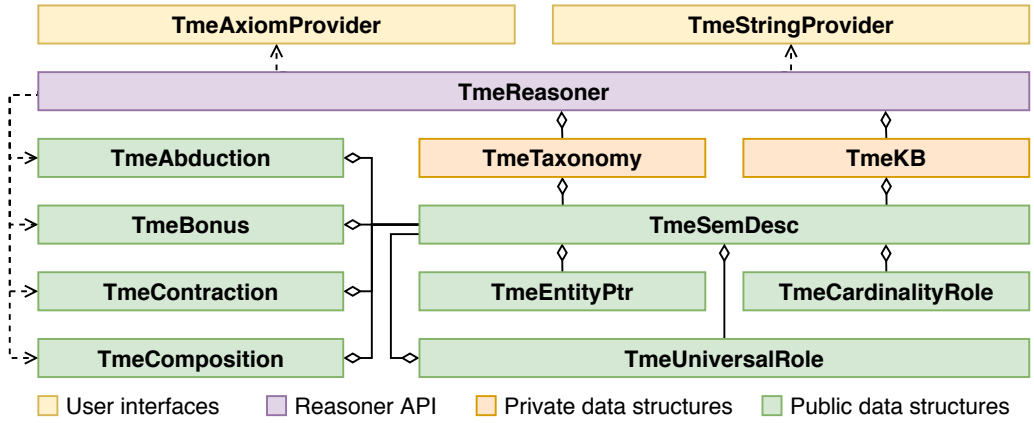


Figure 3.2: Tiny-ME core architecture.

thy that, while C lacks native object-oriented features, it is entirely feasible to construct highly cohesive components by logically bundling structured data with functions that operate on them. This approach has been consistently applied across the core, leading to a codebase that is both highly modular and easily maintainable. The key components of the core architecture are depicted in Figure 3.2 and are 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 writer API.
- `TmeReasoner`: implements reasoning tasks over ontologies, namely Classification and Coherence, 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 [98].

- `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, 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 3.2.

3.4 High-level interaction

As introduced in Section 3.3, the integration of Tiny-ME in a variety of platforms and technological stacks is facilitated by a number of high-level APIs, described in what follows.

3.4.1 Platform-specific APIs

Tiny-ME provides the following platform-specific APIs:

- **C:** the native interface of the system, providing access to the public API of the reasoning core. It achieves this by implementing the axiom and string providers through the *Cowl* library, discussed in Chapter 2. This interface is compatible with any platform that supports a C compiler, making it a highly versatile choice. It is particularly recommended for performance-critical and embedded software development scenarios, as C remains the most widely used language in these areas due to its efficiency and ease of implementation [46].

- **Java:** this interface is tailored for the Java SE and Android runtime environments. It implements axiom and string providers through the OWL API [49]. The data model is essentially a mapping between Java classes and methods to the corresponding C structures and functions. This is achieved via the *Java Native Interface* (JNI).[9] For instance, inference services are accessible through the `Reasoner` class, which encapsulates the native `TmeReasoner` structure and conforms to the OWL API's `OWLReasoner` interface. Class expressions are represented by the `SemanticDescription` class, linking to the native `TmeSemDesc` structure. This pattern is replicated for other logic constructors. Significant attention has been given to the management of native memory: Java objects that are backed by native structures are monitored by the `NativeMemoryManager`, exploiting *phantom references*[10] to track objects on the verge of being garbage-collected, allowing for the timely invocation of native deallocators.

- **Objective-C:** the preferred interface for iOS and macOS applications. Axiom and string providers are implemented via the OWL API for iOS

[9] Java Native Interface: `https://docs.oracle.com/en/java/javase/13/docs/specs/jni/index.html`

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

[96]. As in the Java API, class instances and methods map lower-level C structures and functions, although the wrapping logic is thinner: since Objective-C is an extension of C, it does not require additional interfaces, leading to a simpler architecture and enhanced performance, as detailed in Section 3.6. Memory management in this environment is also more straightforward, as *Automatic Reference Counting* (ARC)[11] and the existence of reliable finalizers[12] enable the synchronization of the lifespan of C allocations with their Objective-C wrappers.

3.4.2 Server-side OWLlink API

To facilitate client-server interactions in Web environments and microservice architectures for cloud and edge computing, Tiny-ME leverages and extends the standard OWLlink protocol [69]. OWLlink provides a declarative interface for OWL reasoners, enabling the assertion of axioms in KBs and the execution of standard inference tasks via standard HTTP requests. Tiny-ME is equipped to handle key reasoning services like Subsumption, Satisfiability, Classification, and Coherence checks. Moreover, it introduces a novel extension to the OWLlink protocol, adhering to the official protocol extension guidelines [68], to incorporate non-standard reasoning capabilities. Drawing from the inference definitions outlined in Section 3.2, new types of requests and their corresponding responses have been defined, as highlighted in Figure 3.3, along with their HTTP/XML binding:

- *GetAbduction, GetBonus, GetDifference*: used to invoke CA, CB, and CD, respectively. These messages require two *ExpressionOrIndividual* arguments, encoding either an OWL class expression or a named individual in the reference KB, corresponding to 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

[11]ARC: `https://clang.llvm.org/docs/AutomaticReferenceCounting.html`
[12]NSObject: `https://developer.apple.com/documentation/objectivec/nsobject`

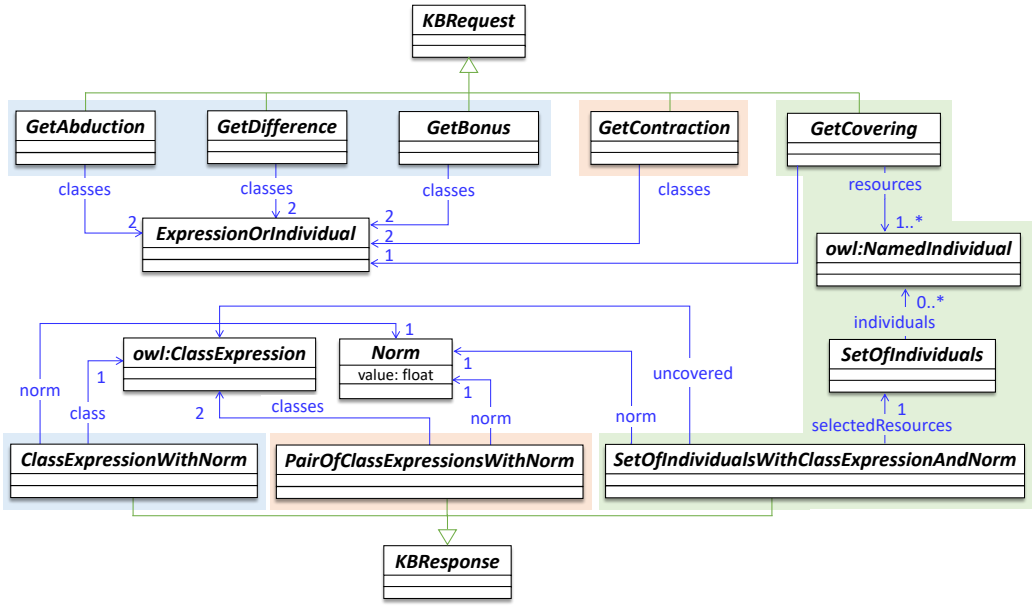


Figure 3.3: OWLlink extension – requests and responses are grouped by color.

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*: the message requires two *ExpressionOrIndividual* arguments modelling a request R and a resource S. The response object consists of two OWL class expressions, corresponding to the conflicting requirements G (Give up) and the contracted (compatible) version K (Keep) of R, with a penalty score measuring the incompatibility degree 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 percentage of R that has not been covered.

A fork of the Java-based *OWLlink API*[13] [81] compatible with OWL API version 5 has been developed to implement the extended interface.[14] 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. `Load Ontologies` requests are supported, instead, for loading a KB from a URL. A *Dockerfile* is available on Tiny-ME's homepage to build a Docker[15] container featuring the reasoner working as an OWLLink server.

3.4.3 Client-side Web API

In order to support client-side reasoning in Web contexts, a port of the system to the WebAssembly runtime has been developed [71], allowing Tiny-ME to run on all major modern desktop and mobile Web browsers. Web developers can use the reasoner through a straightforward JS API, mapping the low-level native API. This has been enabled by the highly portable nature of the system's C core, which allows it to be cross-compiled for the WebAssembly runtime through the *Emscripten*[16] toolchain, and then invoked from JS.

Emscripten does provide mechanisms to call C functions from JS,[17] but they are cumbersome and bug-prone, as they rely either on specifying function signatures via strings, or on manually marshalling parameters to appropriate types. A better alternative involves creating JS wrappers for all exported C functions, though this results in a procedural API, which is rather unnatural for JS development, and leaves most native memory management to the programmer. A third method consists in providing a thin object-oriented C++ wrapping layer and generating bindings to JS classes through *Embind*.[18]

[13] OWLlink adapter: `https://github.com/ignazio1977/owllink-owlapi`

[14] OWLlink matchmaking extension: `https://github.com/sisinflab-swot/owllink-matchmaking-extension`

[15] Docker: `https://www.docker.com/`

[16] Emscripten: `https://emscripten.org`

[17] Connecting C and JavaScript: `https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html`

[18] Embind documentation: `https://emscripten.org/docs/porting/connecting_cpp_and_JS/embind.html`

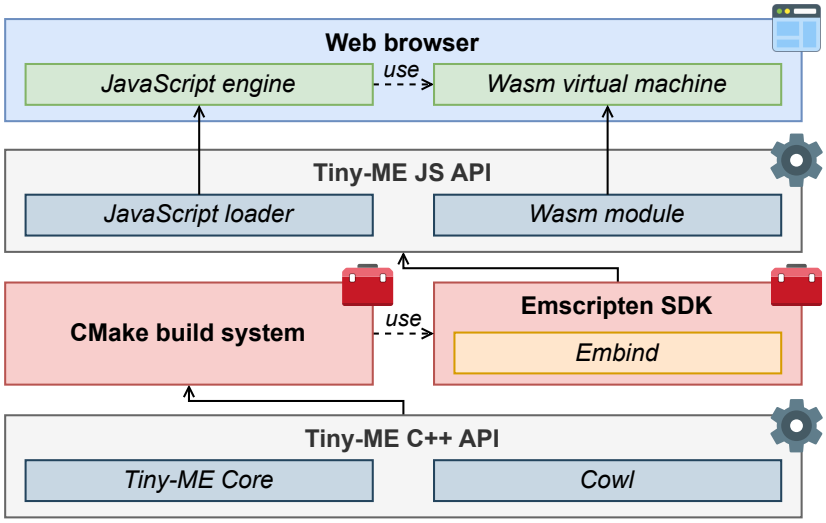


Figure 3.4: Porting workflow and architecture of Tiny-ME for WebAssembly.

This results in a more natural object-oriented JS API, and allows delegating most native memory management to the C++ runtime. Thus, the porting workflow of the reasoner, illustrated in Figure 3.4, has involved the following steps:

- Implementing an object-oriented C++ API by wrapping the Tiny-ME core component and the Cowl library, used by the reasoner to access and parse OWL 2 ontologies.
- Providing appropriate binding annotations, in order to allow Embind to generate a JS interface that directly maps the low-level C++ API.
- Configuring the *CMake*[19] build system to use Emscripten and invoking it to cross-compile the system for the WebAssembly runtime.

Emscripten generates two output artifacts: a Wasm module containing bytecode for the WebAssembly virtual machine, and a JS file responsible for loading the module and setting up the runtime environment. The latter can

[19]CMake home: `https://cmake.org`

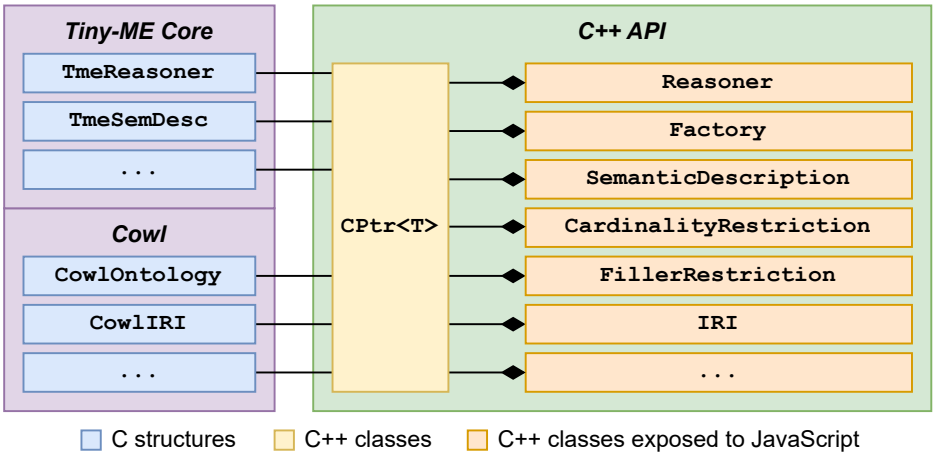


Figure 3.5: Low-level architecture of Tiny-ME for WebAssembly.

be imported in any Web application by means of HTML `script` tags or *JS modules*,[20] allowing client-side code to use the reasoner as a regular JS library.

Figure 3.5 summarizes the low-level system architecture. The object-oriented design of the C API of the reasoner, described in Section 3.3, has resulted in a relatively straightforward mapping of C structures and related functions to C++ classes and methods. One friction point is due to the significantly different memory management paradigms of the underlying libraries, as the Tiny-ME core follows the traditional `malloc`/`free` approach, while Cowl adopts *reference counting*. Care has been devoted to standardizing object lifecycles under the *Resource Acquisition Is Initialization* (*RAII*) paradigm, by encapsulating C pointers in a `CPtr<T>` template type, that implements type-specific memory management logic. This approach simplifies the handling of dynamic memory by delegating it to the C++ runtime, and allows the resulting JS bindings to correctly dispose of unneeded allocations.

Compatibility of the JS API has been tested on the following browser/-platform combinations:

- Chrome (version 90.0.4430) and Firefox (version 88.0.1) on Windows 10 May 2020 Update, Ubuntu 20.10, macOS Big Sur and Android 10;

[20] JS modules: `https://developer.mozilla.org/en-US/docs/Web/JS/Guide/Modules`

- Edge (version 88.0.705.74) on Windows 10;
- Safari (version 14.0) on macOS Big Sur, iPadOS 14 and iOS 14.

Full compatibility has been observed in all tests. The usefulness of the client-side API was demonstrated in [72], where a preliminary WebAssembly port was included in a Web application for semantic-based quality of experience adaptation in Web multimedia streaming. A further usage example will follow in Section 5.4.

3.5 Evolution

The reasoner has undergone significant evolutions since its original 1.0 release [94]. This section provides details about the latest design and development efforts, which have been devoted in three fundamental areas:

- Extending the expressiveness of the provided inference services by providing support for additional DL constructs.
- Improving the flexibility of penalty computation for non-standard inferences, so that they can be tailored for specific use-cases and applications.
- Improving the architecture of the reasoner, and optimizing it so that it can be deployed to increasingly low-footprint platforms.

An evaluation of the latest Tiny-ME version (1.3), with regard to inference correctness and performance, is provided in Section 3.6.3. The updated system is significantly more efficient than the previous iteration with respect to both turnaround time of inferences and memory usage, and is able to run demanding inference tasks, such as ontology classification, on microcontrollers with less than 100 KiB of RAM.

3.5.1 Support for the $\mathcal{ALN}(\mathcal{D})$ DL

The expressiveness of inference services provided by the reasoner has been extended to support the following constructs:

- **Axioms about $\top$:** it is now possible to specify that $\top$ is a subclass of an arbitrary conjunction of constructs, including cardinality and value restrictions. This allows stating ontological knowledge that affects all the concepts and individuals in the KB.
- **Functional roles:** one can now specify that some role R is functional, *i.e.*, that for each individual I_1, there can be at most one distinct individual I_2 such that I_1 is connected by R to I_2.
- **Role ranges:** it is now possible to specify that the range of some role R is an arbitrary class expression C, *i.e.*, if some individual is connected by R with an individual I, then I is an instance of C.

Support for functional roles and role ranges is actually a byproduct of the improved handling of axioms about $\top$, as they are semantically equivalent to $\top \sqsubseteq \leq 1R$, and $\top \sqsubseteq \forall R.C$, respectively. Axioms about $\top$ are supported by explicitly handling the $\top$ concept expression during the unfolding and normalization preprocessing steps: previously, the $\top$ concept expression was ignored by the reasoner as it would result in an infinite unfolding if it contained value restrictions. However, since by definition $\forall C : C \sqsubseteq \top$, which in $\mathcal{ALN}$ entails that the $\top$ concept expression is common to all concept expressions C, it can be *subtracted* through the Concept Difference inference service from all C, as long as its compatibility with C is checked beforehand. The unfolding and normalization procedures can thus be modified as follows: firstly, C is unfolded and normalized as usual; subsequently, C is checked for compatibility with $\top$: if compatibility holds, Concept Difference between C and $\top$ is performed; otherwise, C is collapsed to $\bot$.

It can be trivially proved that the application of the modified unfolding and CNF normalization procedure results in a concept expression that is

equivalent to the original concept expression, which in turn proves that the correctness and completeness of inferences is maintained.

Proof. Let $\mathcal{T}$ be an $\mathcal{ALN}$ TBox; let C and $\top_d$ be arbitrary $\mathcal{ALN}$ concept expressions; let $\mathcal{T} \models \top \sqsubseteq \top_d$, or equivalently $\mathcal{T} \models \top \equiv \top_d$. If $C \sqcap \top_d \sqsubseteq \bot$, then $C \sqcap \top \sqsubseteq \bot$, *i.e.*, C is unsatisfiable. Otherwise, let $C_d = CD(C, \top_d)$. Then, by the definition of CD: $C_d \equiv C \sqcap \top_d \equiv C \sqcap \top \equiv C$. ■

Other than supporting axioms about $\top$, inference services have been extended to concrete domains through OWL datatypes, effectively increasing the expressiveness of the reasoner to the $\mathcal{ALN}(\mathcal{D})$ DL. Tiny-ME adopts a layered approach to concrete domains, where datatypes as defined in the system core have specific semantics, and high-level APIs provide a mapping between OWL and Tiny-ME datatypes. Datatype support, as implemented in the core, is subject to the following semantics:

- **All datatypes are disjoint**, *i.e.*, the reasoner assumes that their intersection is always unsatisfiable. The only exception is `TME_DT_TOP`, the top datatype, which subsumes all other datatypes.
- **Datatypes can be restricted:** it is possible to define *data ranges*, restricting the range of allowed values, by applying type-specific *predicates* to the original datatype.
- **Data ranges do not need to be disjoint:** data ranges obtained by restricting the same datatype do not need to be disjoint, enabling support for datatype subsumption ($D_1 \sqsubseteq D_2$, with D_1 and D_2 data ranges of the same datatype). The reasoner is able to check subsumption between two different datatypes, however in that case it never holds, with the exception of `TME_DT_TOP`.
- **Data ranges can be intersected:** it is possible to combine predicates on the same datatype through the intersection operator ($D_1 \sqcap D_2$, with D_1 and D_2 predicates on the same datatype). The reasoner also allows

intersecting different datatypes, however the result is always `TME_DT_BOTTOM`, the unsatisfiable datatype.

- **Datatypes and predicates can be the fillers of universal quantifiers:** datatype support is plugged into the existing architecture by allowing the fillers of (functional) role value restrictions to be datatypes and predicates ($\forall R.D$, with D being a datatype or predicate).

With respect to OWL 2, support for concrete domains enables working with *datatypes* and *data properties*, *i.e.*, properties that connect individuals to literals. *Data ranges* can also be defined, by restricting the value spaces of datatypes through a restraining *facet*, as per the specification. Furthermore, support was added for *datatype definitions*, which allow defining new datatypes as being semantically equivalent to some data range. Table 3.2 lists all the new OWL constructs supported by the reasoner.

Table 3.2: New supported OWL 2 features in Tiny-ME 1.3.

Axioms	*ObjectPropertyRange, FunctionalObjectProperty, DataPropertyRange, FunctionalDataProperty, DatatypeDefinition*
Datatypes	*rdf:PlainLiteral, xsd:integer, xsd:long, xsd:int, xsd:short, xsd:byte, xsd:negativeInteger, xsd:nonPositiveInteger, xsd:positiveInteger, xsd:nonNegativeInteger, xsd:unsignedLong, xsd:unsignedInt, xsd:unsignedShort, xsd:unsignedByte, xsd:float, xsd:double, xsd:dateTime, xsd:string, xsd:hexBinary, xsd:base64Binary, xsd:anyURI, xsd:boolean, xsd:XMLLiteral*
Data ranges	*DataIntersectionOf, DataAllValuesFrom, DataSomeValuesFrom, DataMinCardinality, DataMaxCardinality, DataExactCardinality, DatatypeRestriction*
Facets	*minInclusive, minExclusive, maxInclusive, maxExclusive, minLength, maxLength, length*

3.5.2 Improved penalty computation

Penalty computation for non-standard inference services can be summarized as in Equation 3.1:

$$penalty(x,y) = N_{xy} + \sum_{i=1}^{M_{xy}} \frac{|r_{xi} - r_{yi}|}{max(r_{xi}, r_{yi})} + \sum_{i=1}^{K_{xy}} \frac{|d_{xi} - d_{yi}|}{2 \cdot max(|d_{xi}|, |d_{yi}|)} + \sum_{i=1}^{L_{xy}} penalty(u_{xi}, u_{yi}) \tag{3.1}$$

with:

- x, y: arbitrary concept expressions (resource and hypothesis for CA, resource and give up for CC).
- N_{xy}: number of atomic and negated concepts that contribute to the penalty.
- M_{xy}: number of cardinality restrictions that contribute to the penalty.
- K_{xy}: number of datatype restrictions that contribute to the penalty.
- L_{xy}: number of value restrictions that contribute to the penalty.
- r_{xi}, r_{yi}: cardinalities of number restrictions that contribute to the penalty.
- d_{xi}, d_{yi}: datatype restrictions (*e.g.*, *minInclusive*, *minExclusive*, ecc.) that contribute to the penalty.
- u_{xi}, u_{yi}: fillers of value restrictions that contribute to the penalty.

The datatype restriction component in the above formula is related to datatypes with number semantics, the only datatypes currently supported by the system. In this case, the penalty score is computed similarly to cardinality restrictions, *i.e.*, as the relative difference of two values ($|x-y|/max(|x|,|y|)$). However, since numerical values for datatypes may have opposite signs, which entails that the relative difference function has a maximum in 2.0 rather than 1.0, the score is computed as $1/2 \cdot (|x-y|/max(|x|,|y|))$.

Other than being rather static, with no way for the user to tailor penalty computation for specific use cases and applications, the current penalty system sometimes leads to undesired outcomes. As an example, if two concepts have different taxonomic depths, then the "deeper" concept usually has a larger effect on the penalty score as a result of concept unfolding, which usually requires careful taxonomy engineering in order to ensure that concepts that may take part in semantic matchmaking are similarly weighted. Another issue

concerns cardinality and datatype restrictions: if the numerical values of the restrictions are large but close, their relative differences are small, therefore they contribute very little to the penalty score. This is problematic because sometimes restriction values may vary in a small range around a common (large) origin, which reduces their relevance when compared with smaller numerical values. Penalty computation has been therefore modified as in Equation 3.2:

$$
\begin{aligned}
penalty(x,y) = & \sum_{i=1}^{N_{xy}} \alpha_{ci} \cdot (1+\beta_{ci}) + \\
& \sum_{i=1}^{M_{xy}} \frac{\alpha_{ri} \cdot |r_{xi} - r_{yi}|}{2 \cdot max(|r_{xi}+\beta_{ri}|, |r_{yi}+\beta_{ri}|)} + \\
& \sum_{i=1}^{K_{xy}} \frac{\alpha_{di} \cdot |d_{xi} - d_{yi}|}{2 \cdot max(|d_{xi}+\beta_{di}|, |d_{yi}+\beta_{di}|)} + \\
& \sum_{i=1}^{L_{xy}} \alpha_{ui} \cdot [\ penalty(u_{xi}, u_{yi}) + \beta_{ui}\ ]
\end{aligned}
\tag{3.2}
$$

Basically, multiplicative factors ($\alpha_i \in \mathbb{R}$, *weights*) and additive factors ($\beta_i \in \mathbb{R}$, *biases*) are added, which are configurable for each class, datatype and role. Appropriate default values can also be set globally. A few considerations are necessary:

- The cardinality restrictions component is modified to be similar to that of the datatype restrictions, as adding the bias can cause the value to become negative.
- The bias does not appear in the numerator of cardinality restrictions and datatypes as it is elided by the difference.
- Biases are unnecessary for atomic concepts, negated concepts, and value restrictions, as they could be absorbed into the weight, however they have been retained for consistency with the other components.

The proposed formula enables the attribution of different weights to concepts, datatypes and roles, and also allows the numerical *origin* of cardinality

Table 3.3: Annotation properties for weights and biases.

Annotation property	Corresponding parameter
swot:conceptWeight	α_c, weight for atomic and negated concepts
swot:conceptBias	β_c, bias for atomic and negated concepts
swot:objectCardinalityWeight	α_r, weight for cardinality restrictions on object properties
swot:objectCardinalityOrigin	$-\beta_r$, origin for cardinality restrictions on object properties
swot:dataCardinalityWeight	α_r, weight for cardinality restrictions on data properties
swot:dataCardinalityOrigin	$-\beta_r$, origin for cardinality restrictions on data properties
swot:dataRestrictionWeight	α_d, weight for value restrictions on data properties
swot:dataRestrictionOrigin	$-\beta_d$, origin for value restrictions on data properties
swot:objectValueWeight	α_u, weight for value restrictions on object properties
swot:objectValueBias	β_u, bias for value restrictions on object properties

and datatype restrictions to be shifted so that they are centered more appropriately depending on the domain of interest. The ability to set different weights for each concept also has the potential to solve the often unwanted asymmetries deriving from concepts with different taxonomic depths. As an example, this system could allow cancelling weights of all the concepts with the exception of "leaf" ones that are intended to contribute to the penalty, which is also simplified by the aforementioned default mechanism. For compatibility with the early formula in Equation 3.1, the following default values are adopted:

- $\alpha_c, \alpha_d, \alpha_u = 1.0$
- $\alpha_r = 2.0$
- $\beta_c, \beta_r, \beta_d, \beta_u = 0.0$

Other than through the reasoner's API, weights and biases can be specified within OWL ontologies through annotation assertion axioms. In this case, the subject of the assertion is the IRI of the targeted OWL entity, the predicate is one of the annotation properties in Table 3.3, whose IRIs fall under the *swot* namespace (`http://swot.sisinflab.poliba.it/owl#`), and the object is a literal that represents the weight or bias. As a special case, the IRI of the subject may be `swot:defaultWeight`, in which case default values are updated.

3.5.3 Updated architecture

Datatype support

The core of the reasoner adopts a *plugin* approach to concrete domains, allowing high-level APIs to define their own datatypes according to a set of pre-defined *semantics*, specifying allowed restrictions and how the datatype is processed during inferences. This system allows defining multiple disjoint datatypes that share the same semantics, *i.e.*, that can be processed in an algorithmically analogous way. Furthermore, implementing new datatype semantics can be easily achieved by providing a set of primitives that perform basic operations on data ranges, such as their intersection, containment, difference, and so on. This extensible system may allow the reasoner to support custom datatypes beyond those defined by OWL 2.

Datatypes are represented by `TmeDatatype`, an integer type that allows high-level APIs to define supported datatypes and their semantics via the `tme_datatype_def` function. Invoking the function generates a new datatype, whose bits are the encoding of:

1. A `TmeDataSemantics` enumeration, used by the reasoner to select the set of manipulation primitives to use when handling the datatype. The values the enum can take are currently: `TME_DS_NONE` for datatypes that do not provide restrictions, and `TME_DS_INT` or `TME_DS_FLOAT` for datatypes with integer and floating point semantics, respectively.
2. An ascending ordinal, that allows distinguishing one datatype from another.

Data ranges are encapsulated within the `TmeDataRange` structure, comprising a `TmeDatatype` and an optional restriction field. This field can be either `TmeIntRestr` or `TmeFloatRestr`, denoting minimum and maximum value restrictions for integer and floating-point data types, respectively. Data property value restrictions (*DataAllValuesFrom* in OWL 2) are finally mapped

into the `TmeDataRestr` structure which, analogously to object property value restrictions, contains a `TmeEntity` identifier for the data property, and a `TmeDataRestr` filler. Data property value restrictions have been exposed in `TmeSemDesc` through a dedicated field. Regarding data property cardinality restrictions, neither `TmeSemDesc` modifications nor inference level changes have been required. The existing data structures are sufficient to represent the corresponding OWL constructs, now supporting restrictions on both object and data properties.

These changes have enabled supporting datatypes such as integers, floats, doubles, booleans, etc. but also strings, dates and URIs, operating as follows:

1. Support for the *pattern* facet has been discarded at the moment, due to complexity and code size concerns: finding out whether a regular expression is a subset of another one would require the inclusion of a regular expression library that supports set operations, which would significantly increase the overall size of the library.

2. The remaining facets required by the specification (*minInclusive*, *minExclusive*, *maxInclusive*, *maxExclusive*, *minLength*, *maxLength* and *length*) have been mapped to the `TmeIntRestr` and `TmeFloatRestr` structures.

3. For datatypes whose literal representation is different from that of an integer (e.g. *xsd:dateTime*), a conversion mechanism from literal to integer and vice versa has been implemented. In the case of dates, the date literal is converted into Unix time,[21] allowing the reasoner to treat it as an integer datatype, and then converted back to a string for visualization purposes.

Following this strategy, support for many of the OWL 2 datatypes has been provided in a relatively non-invasive and lightweight manner, with enough flexibility to easily allow support for additional datatypes.

[21] Computed as the seconds elapsed since midnight, January 1, 1970.

Penalty computation

The enhancements to penalty computation described in Section 3.5.2 are facilitated by the `TmeWeightMap` data structure and its API, which supports setting and retrieving weights and biases used to control the computation of penalty scores of non-standard inference services, as described in Section 3.5.2.

Other than setting weights and biases programmatically, they can be automatically parsed from annotations in the source ontology, in which case annotation axioms are processed alongside logical axioms when instantiating the reasoner, and the `TmeWeightMap` structure is updated accordingly.

Optimizations

The updated system implements a plethora of architectural and low-level optimizations, which together enable its deployment to the very low end of the SWoE device spectrum, as will be showcased in Section 3.6. The most relevant optimizations are reported hereafter.

- Tiny-ME is now built on top of the *uLib* library, which acts as the foundation of the whole KRR infrastructure, providing highly optimized data structures that are shared between Cowl and the Core component, allowing for a reduced code section when the two components are used simultaneously, such as when using the C API of the system.
- The axiom streams feature of the Cowl library has been adopted by the Tiny-ME C API. In the previous iterations, the ontology document was first parsed into a *CowlOntology* data store, which was later queried in order to populate the reasoner's internal data structures. Version 1.3 parses the ontology as an axiom stream (see Section 2.4), and each axiom is translated and added to the reasoner's internal knowledge base on the fly. This significantly lowers the memory peak during reasoner initialization, as the axiom store and the reasoner are never simultaneously in memory, and allows skipping the processing of unsupported axioms and other extra-logical constructs such as annotations.

- The most performance-critical data structures have been further optimized, focusing on memory usage and computation time. As an example, the `TmeSemDesc` structure, representing $\mathcal{ALN}$ concept expressions, has been redesigned so that its baseline memory usage is lower, and it takes up progressively more memory the more construct types it contains. Furthermore, vectors of constructs are kept in sorted order, so that queries can occur in $O(log(N))$ time via binary search, while still keeping a compact in-memory representation which would be lost by using alternate data structures such as hash tables or trees. This of course entails that insertion and removal of constructs happen in $O(Nlog(N))$ time, though it still results in an overall performance gain as the data structures are read much more often than they are modified. Finally, the structure now internally tracks its unfolding and normalization state, eliminating the need to maintain a dedicated cache. This saves memory, due to the absence of a separate data structure, but also time, as checking whether the description needs to be unfolded or normalized can be done by accessing an internal field rather than through a much more expensive hash table lookup. The `TmeTaxonomy` data structure has been also optimized, by reducing the baseline memory usage of each node and removing the *subsumption cache*, obsoleted by the much improved performance of `TmeSemDesc` during queries.

3.6 Evaluation

This section reports the results of a number of experimental campaigns carried out with Tiny-ME on various reference testbeds and inference tasks, with the goal to validate its adaptability to different technological and resource constraints, and to compare it to other state-of-the-art reasoning systems.

3.6.1 Workstation and mobile

The first experimental campaign has been carried out in [94] to evaluate the computational performance of Tiny-ME 1.0 in Ontology Classification and non-standard inference tasks. Tests have been executed on small workstation and mobile platforms by means of the evOWLuator framework (see Chapter 4). The workstation testbed is an *Apple Mac Mini (2014)*[22], while mobile tests have been carried out on an *Apple iPhone 7*[23] and a *HTC/Google Nexus 9* tablet[24].

The dataset exploited for the classification test consists of 1364 knowledge bases obtained from the 2014 *OWL Reasoner Evaluation Workshop* competition[25] out of the 16555 KBs in the DL classification corpus (8.24%), considering only KBs having at most $\mathcal{ALN}$ as indicated expressiveness. As said, tests refer to both workstation and mobile platforms.

Correctness evaluation has involved comparing the outputs of the system with Konclude and Mini-ME Swift, used as test oracles respectively for Ontology Classification and the non-standard Matchmaking task. All Tiny-ME variants have provided correct and complete inferences for all supported ontologies and reasoning tasks.

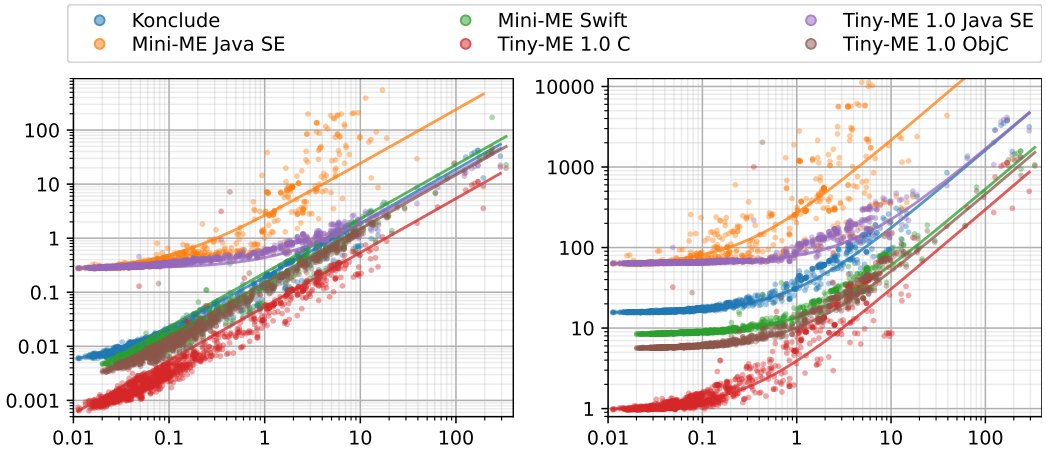
In what follows, all performance results are the average of five cold runs. *Peak memory usage* refers to the *maximum resident set size* (MRSS) of the reasoner process, measured by means of the `getrusage` POSIX call on iOS and Android, and by evOWLuator itself on desktop/workstation platforms. Standard 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).

[22]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

[23]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

[24]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

[25]ORE 2014 corpus: `http://dl.kr.org/ore2014`



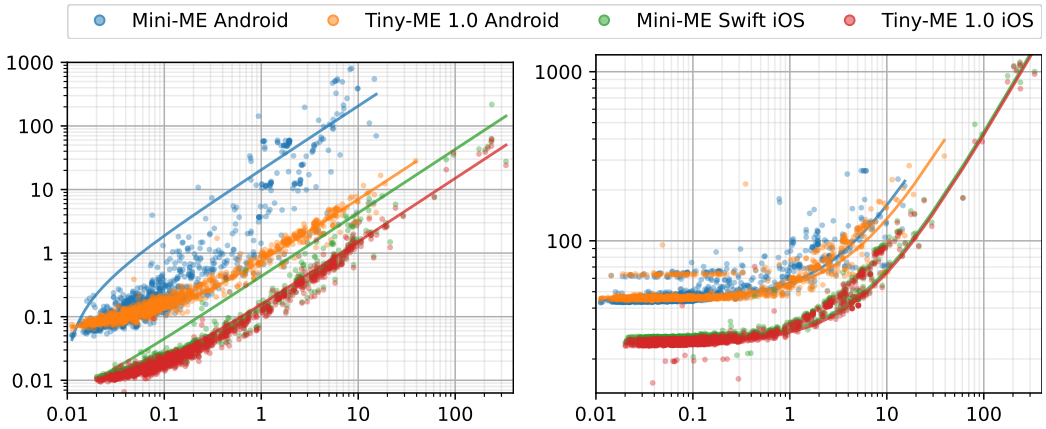
(a) Time (s) by ontology size (MiB). **(b)** Memory peak (MiB) by ontology size (MiB).

Figure 3.6: Comparison of classification time and memory peak on workstation.

Konclude has been considered as reference reasoner, due to previous campaigns [84, 98] 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 system. Figures 3.6a and 3.6b depict inference turnaround time and memory peak as a function of ontology size. The comparison involves the Tiny-ME C, Java SE and Objective-C (ObjC) platform-specific interfaces, Mini-ME 2.0 for Java SE, and Mini-ME Swift for macOS.

Aggregated performance metrics, such as those reported in Table 3.4, refer to the set of 1168 ontologies that all reasoners could classify correctly within a 20 minutes 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 3.4, Tiny-ME C has outperformed all the other systems with respect to both time and memory usage. The gap is particularly



(a) Time (s) by ontology size (MiB). **(b)** Memory peak (MiB) by ontology size (MiB).

Figure 3.7: Comparison of classification time and memory peak on mobile.

evident for ontologies smaller than 1 MiB, which is a relevant outcome for resource-constrained SWoE scenarios.

The Objective-C API exhibits lower times and memory usage than the Java one, as expected due to thinner Objective-C wrappers than those necessary to interface with the JNI, as explained in Section 3.4.1. 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 show similar trends in terms of processing time and memory occupancy for small ontologies, but they diverge when the input size grows (see Table 3.4). 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 3.4.

Figure 3.7 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 on the workstation tests. Reasoners running on iOS have been able to process all the ontologies in the dataset within the imposed timeout and without errors,

Table 3.4: Dataset-wide evaluation results for classification.

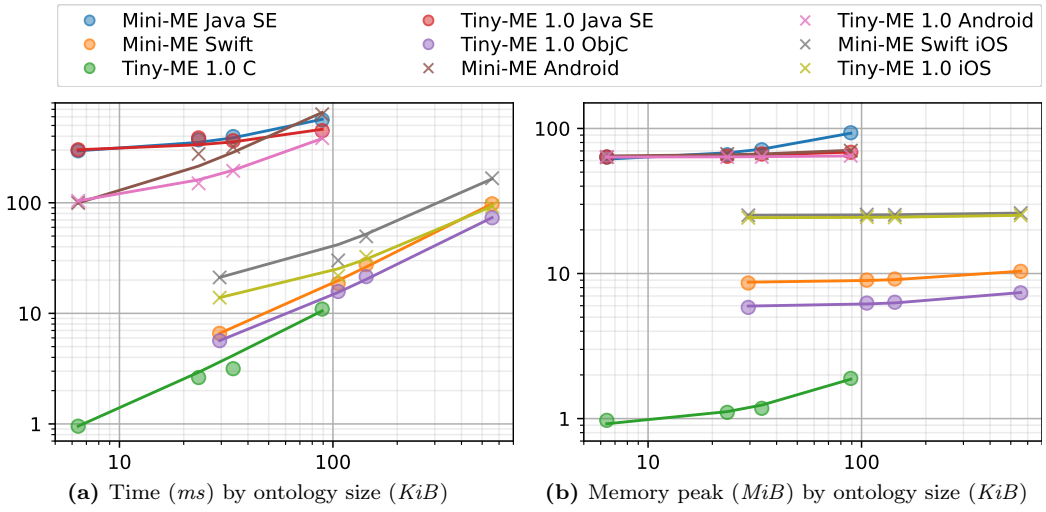
Platforms	Reasoners	Errors and Timeouts	Parsing Time (s)	Classification Time (s)	Minimum Memory Peak (MiB)	Maximum Memory Peak (MiB)
Workstation	**Konclude**	0	82.16	64.37	15.53	1559.52
	Mini-ME Java SE	196	443.01	4516.83	62.49	11237.72
	Mini-ME Swift	0	131.80	30.08	8.41	1065.92
	Tiny-ME 1.0 C	0	27.26	8.41	0.96	307.55
	Tiny-ME 1.0 Java SE	0	447.96	67.02	62.98	1408.6
	Tiny-ME 1.0 ObjC	0	131.14	12.31	5.62	1032.87
Mobile platforms	**Mini-ME Android**	237	326.71	8084.61	43.09	259.88
	Tiny-ME 1.0 Android	9	343.05	96.54	44.18	133.71
	Mini-ME Swift iOS	0	450.07	344.95	24.94	1140.18
	Tiny-ME ObjC iOS	0	446.41	113.68	23.95	1117.37

Table 3.5: Features of KBs used for non-standard inference tests.

Knowledge Base	Toy	Agriculture	Building	MatchAndDate
Size (KiB)	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

while Android reasoners have started running out of memory while classifying ontologies larger than about 20 MiB. This issue is evident in both Figure 3.7, where missing data points are observable for larger ontologies on Android, and Table 3.4, which shows lower maximum memory peaks for Android reasoners compared to their iOS counterparts, attributable to memory exhaustion errors.

All Tiny-ME variants have been also evaluated on non-standard inference tasks, using four $\mathcal{ALN}$ KBs summarized in Table 3.5. Following the approach in [98], a matchmaking task starts with a *compatibility* check be-



(a) Time (ms) by ontology size (KiB)

(b) Memory peak (MiB) by ontology size (KiB)

Figure 3.8: Overall processing time and memory peak for the matchmaking task.

tween a $\langle request,\ resource \rangle$ pair: Concept Abduction is performed in case their conjunction is satisfiable, otherwise Concept Contraction is executed, followed by Concept Abduction on the contracted version of the request. This corresponds to the matchmaking scheme outlined in Section 1.2.2, bar the final Concept Bonus calculation.

Figure 3.8 recalls main outcomes. It should be noted that ontology size varies between 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 behavior, and they are both more efficient than Java-based implementations. Figure 3.8a highlights lower turnaround times on Android than on Java SE, as the overall time includes parsing, which is slower on the Mac Mini compared to the Nexus 9 (the latter has faster mass storage). Figure 3.8b 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 on mobile, as a consequence of stricter resource management policies. The opposite trend is observed on iOS because, as stated in [98], iOS reasoner variants are affected by a systematic memory

overhead due to their graphical user interface, while the other ones can run as command-line tools.

3.6.2 Client-side WebAssembly

An experimental campaign has been conducted [71] in order to evaluate the performance of the WebAssembly port of the reasoner in terms of turnaround time of standard and non-standard inference services. For each test, the following metrics have been measured:

- **Fetching**: the time spent by the client to request and download the target ontology. This value does not depend on the reasoner, but only on the device, the ontology size, and the network link between the client and the server.
- **Parsing**: the time required by the reasoner to deserialize the ontology into its internal data structures.
- **Reasoning**: the time elapsed to carry out the requested inference service.

Tests have been executed on a 2021 MacBook Pro,[26] an iPhone 12 Pro,[27] and a OnePlus 7T,[28] to achieve a sufficiently fair and diverse representation of Web content fruition devices, both hardware- and software-wise.

The reference datasets are the same as those used in Section 3.6.1, but considering only ontologies whose file size does not exceed 1 MiB in OWL 2

[26] Apple M1 Max SoC, 8 performance cores @3.2GHz and 2 efficiency cores @2.0GHz, 64 GB UM RAM, 1 TB SSD, macOS Ventura 13.3.1, Safari 16.4 browser.

[27] Apple A14 Bionic, 2 high-power cores @3.1 GHz and 4 low-power cores @1.8 GHz, 6 GB LPDDR4X RAM at 4266 MT/s, 128 GB NVMe SSD, iOS 15.1, Safari Mobile 15.1 browser.

[28] Qualcomm Snapdragon 855+, 1 core @2.96 GHz, 3 cores @2.42 GHz and 4 cores @1.8 GHz, 8 GB LPDDR4X RAM, 128 GB UFS 3.0 storage, Android 12, Chrome 112.0.5615.47 browser

functional-style syntax [85]. The classification corpus is thus reduced to 1140 ontologies. This does not affect the dataset used for non-standard tests.

The experimental method consists in visiting a Web page containing client-side JS code responsible for performing all test operations. For each ontology in the dataset, the script retrieves the ontology document, then invokes the Tiny-ME JS API to carry out the target inference. Results are saved to a Comma-Separated Values (CSV) file, which is then fed to the EVOWLUATOR framework to generate visualizations (see Section 4.2.3). The test Web pages and ontologies are served by an *NGINX*[29] instance hosted on a desktop computer,[30] configured to serve all requests without caching. All devices are located in the same IEEE 802.11n WLAN.[31]

Times are collected using the `performance.now()`[32] JS API call. By default, most browsers limit the resolution of the returned timestamp to 1-2 milliseconds as a mitigation for timing-based attacks and fingerprinting. This is enough for fetching and parsing, but it is often too coarse for inferences. Therefore, reasoning times have been computed by running multiple consecutive iterations of each task (10 for the two mobile devices, and 50 for the laptop), subtracting the cumulative fetching and parsing times from the total execution time, and averaging over all iterations.

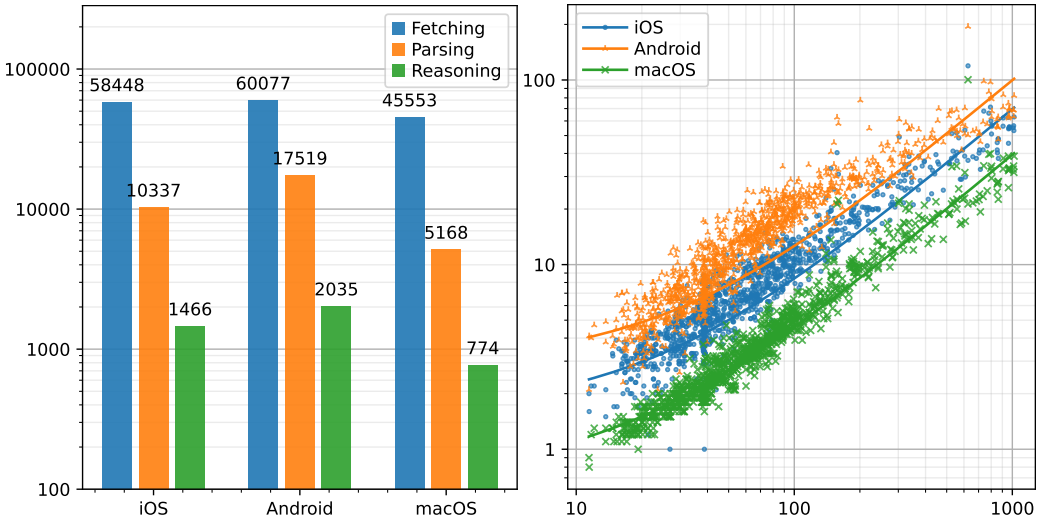
Figure 3.9 illustrates the results of the *ontology classification* standard inference service. The bar graph in Figure 3.9a shows the total time required for fetching, parsing, and reasoning across all ontologies. The scatter plot in Figure 3.9b depicts the relationship between the size of the ontology and the total time required for parsing and reasoning, with the worst case requiring about 200 ms on the least capable device. Outcomes demonstrate satisfactory performance of inference services oriented to ontology management, which is essential for supporting interactive semantic-enabled Web applications in mobile environments. Although fetching time is the most prominent

[29] `https://www.nginx.com/`

[30] Intel Core i7-3770k CPU, 4 cores @3.5GHz, 12 GB DDR3 RAM @1600 MT/s, 2 TB SATA SSD, Windows 10.

[31] Hosted by a TP-Link TN-WR841N router.

[32] `https://developer.mozilla.org/en-US/docs/Web/API/Performance/now`



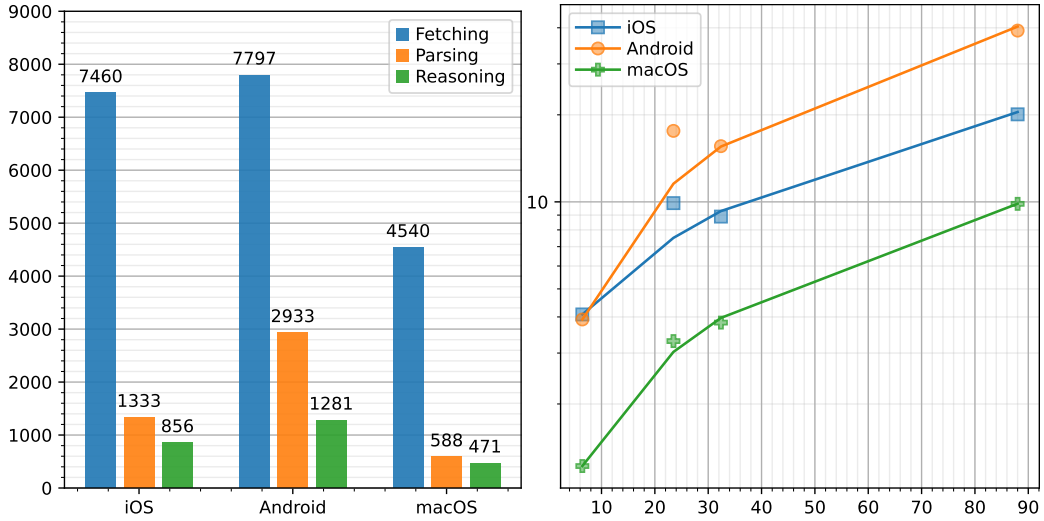
(a) Cumulative Reasoning Time (ms). **(b)** Reasoning Time (ms) by ontology size (KiB).

Figure 3.9: Ontology classification results for the JS API.

component, the domain ontology can usually be fetched just once and cached to improve application responsiveness.

Figure 3.10 depicts performance metrics for the *matchmaking* non-standard service. The Wasm reasoner can carry out this inference in a few milliseconds, with 40 ms being the highest time measured for the largest ontology of the corpus. These outcomes align with the ones for Ontology Classification, but they are particularly relevant, as non-standard inferences are often more useful in SWoE scenarios, which Tiny-ME explicitly targets.

A preliminary matchmaking memory usage test has been carried out on the aforementioned MacBook Pro testbed, by profiling the system via the snapshotting capabilities embedded into the Mozilla Firefox browser developer tools [72]. By default, the Emscripten compiler statically allocates a 16 MiB contiguous chunk of memory for the Wasm module, which is problematic because it does not allow to know how much memory is actually needed at runtime by the matchmaking task. Therefore the initially allocated memory has been decreased to 192 KiB, the minimum amount for which the code would compile, and the Wasm module has been recompiled with the `ALLOW_-`



(a) Cumulative Reasoning Time (*ms*). **(b)** Reasoning Time (*ms*) by ontology size (*KiB*).

Figure 3.10: Semantic matchmaking results for the JS API.

`MEMORY_GROWTH` flag[33] enabled, which lets the browser allocate additional memory for the module if necessary. Given the above configuration, the following memory snapshots have been taken:

1. *Baseline:* blank page, before loading the Wasm module.
2. *Wasm module:* after loading the Wasm module.
3. *Runtime:* memory usage peak, while matchmaking.

Loading the Wasm module introduces a 1.16 MiB memory overhead over the baseline, which includes the Wasm memory buffer (initially sized at 192 KiB, as said). While matchmaking, an additional maximum of 0.31 MiB of memory is allocated, and the Wasm memory buffer is dynamically resized by the browser to 320 KiB. The results evidence a tolerable memory overhead for the matchmaking process, corroborating the feasibility of the proposed approach.

The overall evaluation confirms that the JavaScript API of the system can be effectively integrated into interactive Web applications, providing standard

[33] `https://emscripten.org/docs/optimizing/Optimizing-Code.html`

and non-standard client-side inference services without negative impact on performance and user experience.

3.6.3 Evolution

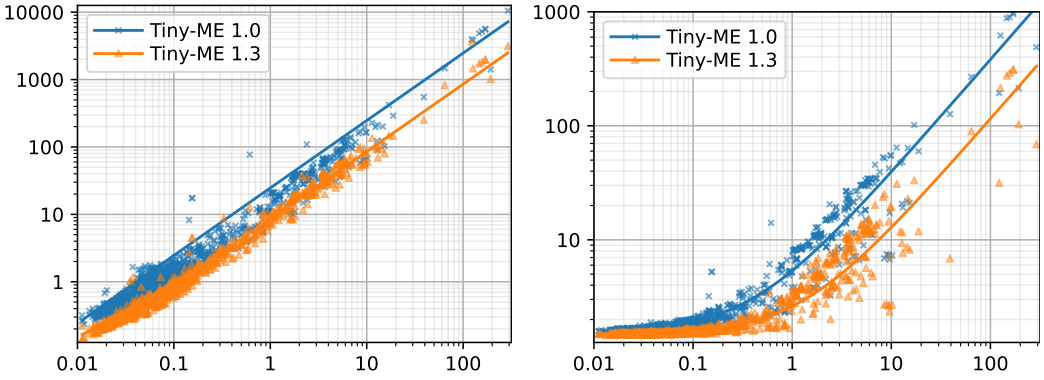
A third evaluation has concerned the updated system, as described in Section 3.5. Tests have been carried out to assess the correctness and completeness of inference services, which have been extended to the $\mathcal{ALN}(\mathcal{D})$ DL, and to determine the effect of the optimizations reported in Section 3.5.3.

Workstation trials have been carried out on a 2021 Apple MacBook Pro 16".[34] Comparative performance tests between Tiny-ME versions 1.3 and 1.0 have used the same dataset as in Section 3.6.1, in order to allow for a direct comparison. All other tests have been executed on a subset of the ORE 2014 dataset, limited to ontologies having $\mathcal{ALN}(\mathcal{D})$ expressiveness at most, *i.e.*, a superset of the previous dataset which also includes ontologies with datatypes. This new dataset consists of 1498 ontologies, of which 134 contain datatypes.

Classification correctness tests have used HermiT [38] as a test oracle. HermiT has been preferred to Konclude in this case as it supports all OWL 2 datatypes, while Konclude implements only part of the OWL 2 datatype map.[35] Tiny-ME has returned correct and complete results for all supported ontologies in the ORE 2014 $\mathcal{ALN}(\mathcal{D})$ dataset. With respect to matchmaking, since there is no other system that supports non-standard inferences on the $\mathcal{ALN}(\mathcal{D})$ DL, inference correctness and completeness have been determined by constructing a test ontology and verifying the results manually.

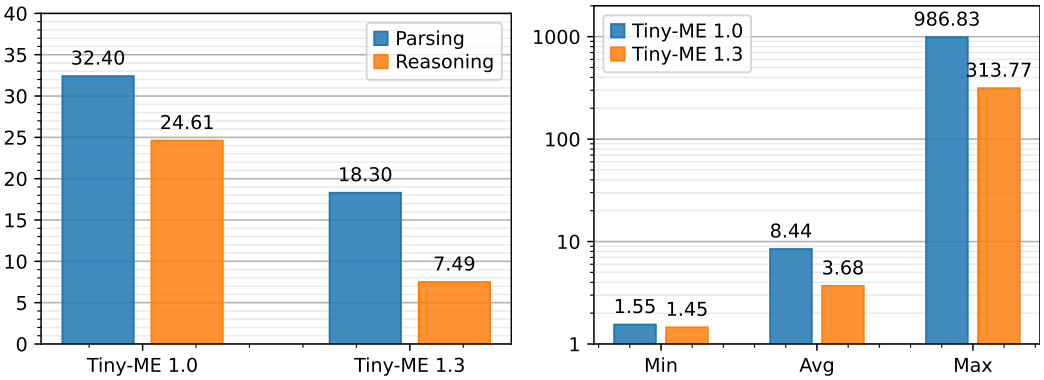
Results of comparative performance tests between Tiny-ME 1.3 and Tiny-ME 1.0 are in Figure 3.11 and Figure 3.12. The updated system significantly outperforms the earlier version when running the ontology classification inference service: cumulative reasoning time is 55% lower, while the average and maximum memory peaks are 56% and 68% lower, respectively. Similar

[34]Apple M1 Max System-on-Chip with 64 GB RAM, 1 TB SSD, macOS Sonoma 14.1.2.
[35]As reported on its public source repository: `https://github.com/konclude/Konclude`



(a) Time (ms) by ontology size (MiB). **(b)** Memory peak (MiB) by ontology size (MiB).

Figure 3.11: Classification performance metrics on workstation.

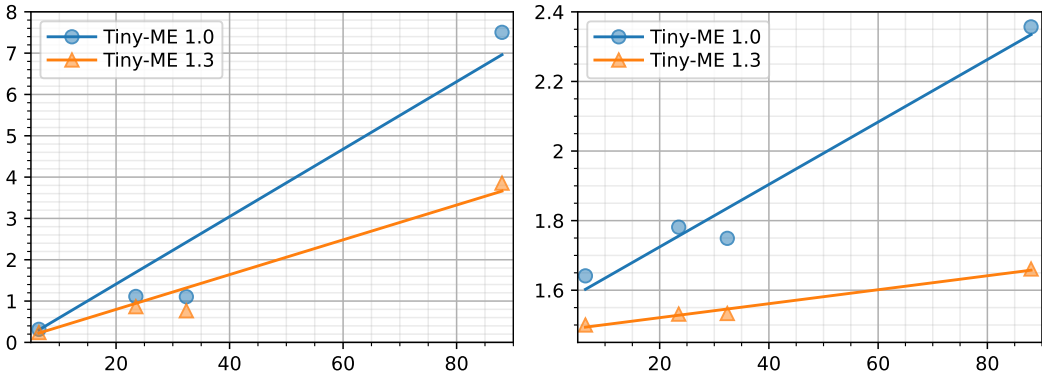


(a) Cumulative time (s). **(b)** Dataset-wide memory peak (MiB).

Figure 3.12: Dataset-wide classification performance metrics on workstation.

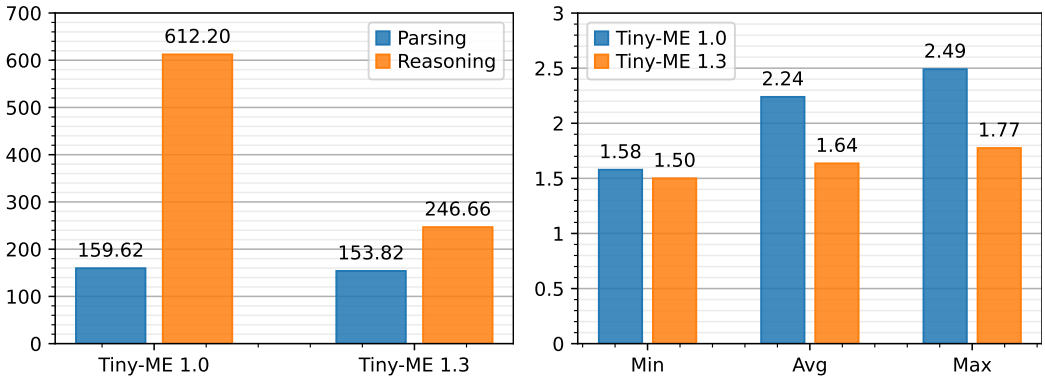
considerations apply for the matchmaking task, as shown in Figure 3.13 and Figure 3.14: cumulative reasoning time is 48% lower, while average and maximum memory peaks are 27% and 29% lower, respectively. These results validate the impact of the significant architectural and performance improvements of the updated system. The substantial memory usage reduction is mainly due to the adoption of the axiom streams technique for ontology parsing, as allowed by the Cowl library, and to the removal of the unfolding and subsumption caches, enabled by the architectural updates to the `TmeSemDesc` core data structure. The latter also brings about improvements in reasoning time for classification and matchmaking, which are similar in magnitude, underlining the performance importance of that data structure.

To really prove that Tiny-ME 1.3 is now able to power a larger band of



(a) Time (ms) by ontology size (KiB). **(b)** Memory (MiB) by ontology size (KiB).

Figure 3.13: Matchmaking performance metrics on desktop.

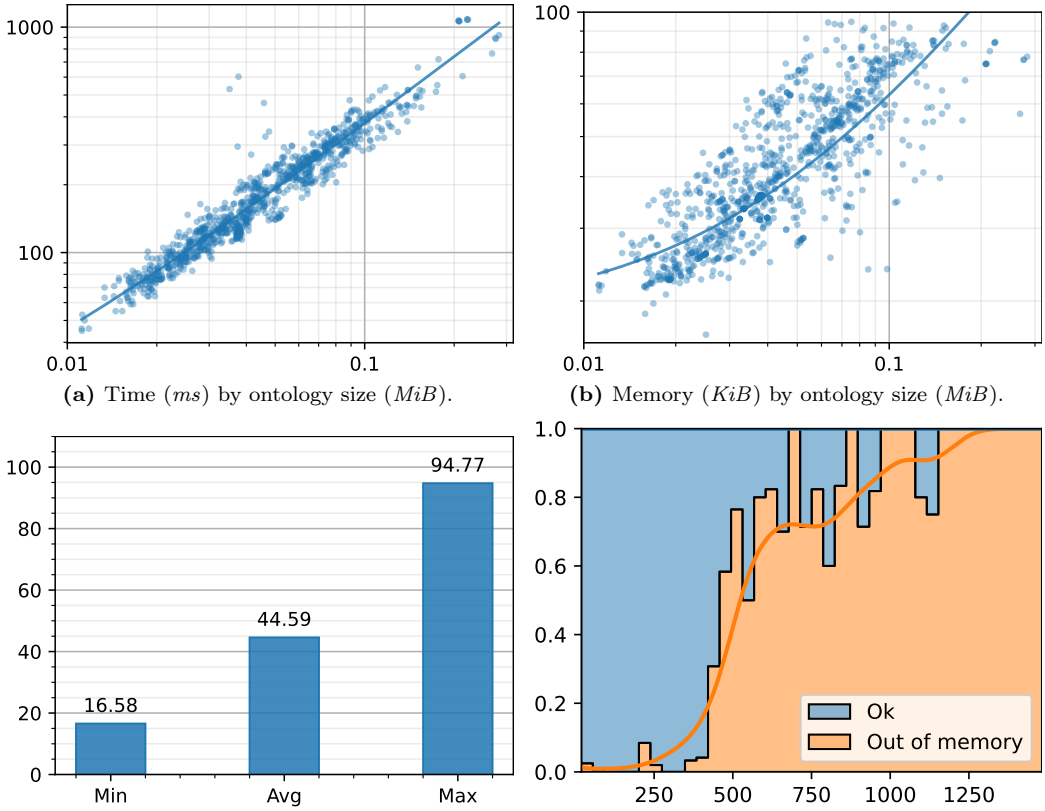


(a) Cumulative time (s). **(b)** Dataset-wide memory peak (MiB).

Figure 3.14: Dataset-wide matchmaking performance metrics on desktop.

the SWoE device spectrum, it has been deployed and tested on an Arduino Due board, using the same configuration as in Section 2.6. The ontology classification reasoning task has been chosen as a benchmark, as it is one of the most demanding inference services for OWL reasoners. Each inference request is carried out by instantiating the reasoner on the inbound USB byte stream, which is transformed into an axiom stream by the Cowl stream-based parser. Each axiom is converted into a suitable internal data model representation and added to the reasoner core data structures. Once the ontology is fully loaded, the reasoner performs classification, building the complete concept taxonomy, and reporting performance metrics to EVOWLUATOR through the UART port.

Figure 3.15 displays the time and memory required to parse and classify



(a) Time (ms) by ontology size (MiB).

(b) Memory (KiB) by ontology size (MiB).

(c) Dataset-wide min, avg, and max memory (KiB).

(d) Ratio of out-of-memory events by axiom count.

Figure 3.15: Classification performance metrics on Arduino Due.

ontologies as a function of their size. Tiny-ME has been able to classify 1006 ontologies (67.2% of the aforementioned dataset), with the largest ontology being 290 KiB, using 44.59 KiB of RAM on average, *i.e.*, about half of the available RAM on the Arduino Due board. The remaining KBs could not be processed due to memory exhaustion, whose trend is displayed in Figure 3.15d as the histogram plot of ontologies grouped by processed axiom count (x-axis) over the percentage of ontologies that result in an out-of-memory (OOM) event within each bin (y-axis). The plot shows the board is able to handle up to 400 axioms without significant issues, while the ratio of OOM events quickly increases after this threshold, reaching 0.8 around 500 axioms. When compared with Figure 2.10a, it is evident that the reasoning process increases memory requirements significantly w.r.t. pure parsing, as the 0.8 mark is reached around 900 axioms when just loading and indexing axioms in the

CowlOntology store.

Overall, system performance aligns well with the requirements of the SWoE, as it demonstrates the capability to perform complex inferences on moderately articulated knowledge bases, making it viable for real world applications. The system's ability to classify a significant portion of ontologies, especially considering the size and memory limitations of the Arduino Due board, underscores its practical utility, proving that even with the inherent challenges of limited resources, Tiny-ME can effectively support key reasoning tasks on severely constrained devices.

Chapter 4

evOWLuator: multiplatform benchmarking for OWL toolkits

This chapter presents evOWLuator [104], a multi-platform framework for the evaluation of OWL reasoners. It is is characterized by high flexibility, expandability, and scalability, achieved through unique architectural choices which set it apart from the state of the art. evOWLuator runs on GNU/Linux, macOS, and Windows (through the *Windows Subsystem for Linux*, WSL), can support any ontology corpus, and is able to evaluate common reasoning services, such as ontology consistency and classification, as well as non-standard ones like semantic matchmaking. Moreover, the set of supported inference services can be expanded by the user through a plug-in mechanism.

One of the key features of the framework is its ability to deploy tests either locally or on remote devices, allowing for integration with mobile and embedded platforms, as demonstrated in the evaluation campaigns reported in the previous chapters. The tool is capable of evaluating various reasoning metrics, including correctness, turnaround time, memory usage, and energy footprint, a first in the OWL benchmarking landscape. evOWLuator's plug-in architecture expands its usefulness by supporting the integration of additional target reasoners, platforms, and reasoning tasks. It can also generate interactive visualizations of results with highly customizable plots, making it a valuable tool for research activities. In order to promote its

adoption in both academic and industrial contexts, its source code[1] is released under a very permissive, commercially-friendly license.[2]

The remainder of the chapter is as follows: Section 4.1 discusses the state of the art; Section 4.2 shows how evOWLuator can be used to carry out experiments and visualize results; Section 4.3 describes the architecture of the framework, with user-configurable interfaces detailed in Section 4.4; finally, Section 4.5 reports the results of a small experimental campaign comparing six OWL reasoners, demonstrating the effectiveness of the approach and tool.

4.1 Background

When looking for the best tool for a particular application, besides functional requirements and platform compatibility, quantitative systematic analysis of performance and scalability becomes crucial. The selection of software components should rely as much as possible on rational processes based on quantitative data, so as to prevent incorrect strategic decisions [139]. This is particularly true when dealing with performance and resource consumption evaluations, which may prevent running on a particular platform or implementing certain desired functionalities and thus lead to reduced acceptability and adoption of products and services.

In the field of Semantic Web technologies, this has motivated the creation of several OWL reasoning benchmarks and automated evaluation frameworks. Selecting an evaluation tool is by itself a non-trivial problem, depending on features like the types of collected performance metrics, platform compatibility, supported inference services, ease of reasoner integration, test automation capabilities, and so on. Furthermore, in latest years, the rise to prominence of mobile and pervasive computing has extended the field of application of knowledge representation and reasoning to non-conventional contexts.

[1] evOWLuator source code: `https://github.com/sisinflab-swot/evowluator`
[2] Eclipse Public License 2.0: `https://www.eclipse.org/legal/epl-2.0/`

Until around 2010, reasoner evaluations primarily relied on benchmarks with a limited number of ontologies and small, handcrafted query sets [51]. A key example is the *Lehigh University Benchmark (LUBM)* [44], notable for its single ontology focused on the university domain, fourteen extensional queries for various properties, and a data generator for scalable ABoxes. LUBM, alongside three additional ontologies and query sets covering four OWL fragments, was used in [19] to assess five reasoners, determining the most suitable ones for each ontology class and inference task. Similarly, [29] compared eight reasoners using three extensive ontologies from the OWL 2 EL profile, focusing on classification, consistency, concept satisfiability, and subsumption checks. Additionally, [80] enhanced LUBM to support SPARQL-based stream reasoning.

With the increased availability of knowledge bases and graphs, recent years have seen the development of larger and more varied corpora for OWL reasoner benchmarking to test systems in real-world scenarios. A 2012 study [55] set a first record by measuring the classification time of four reasoners on a dataset of over 300 real-world ontologies, also using ontology metrics as machine learning features to predict processing time. The *OWL Reasoner Evaluation* (ORE) workshop series, running annually from 2012 to 2016, expanded the scope with more ontologies, reasoning tasks, and participants [39, 84]. For each reasoner, a score was determined by the number of problems solved out of the total in each competition track, and time was used to break ties for the final standings. Unfortunately, other performance indexes like memory or energy usage were not taken into account.

The growth of datasets and test cases underscored the need for automated benchmarking tools, with the framework from [84] being a notable solution for traditional computing platforms. Before starting the evOWLuator project, adapting that tool to support additional metrics and platforms was considered, but it was ultimately deemed too complex, as the framework was primarily for live competitions and focused on inference correctness. Extending it would have implied significant additions to an already large project (over 200 Java files with a total of 15000 lines of code, excluding the Web interface),

which was not designed for such expandability or for the inclusion of features like memory and energy evaluation or mobile platform support. In contrast, evOWLuator offers a more compact and flexible solution, with only about 4000 lines of code organized in about 40 Python files, supporting several types of customizations through plug-ins without altering the existing codebase.

Specialized evaluation frameworks exist for testing non-standard inferences. *JustBench* [7] focuses on assessing reasoner performance in verifying *justifications*, minimal subsets of an ontology required for an entailment. Performance evaluations for Mini-ME in Java/Android [107] and its Swift reengineering [98] for iOS focused on the matchmaking task. With the growing application of semantic technologies in ubiquitous computing, there is an increased emphasis on benchmarking mobile-specific OWL profiles and emerging mobile reasoners. A notable study [18] tested six reasoners with OWL API [49] support on Android, using the ORE 2013 dataset [39], focusing on ontology classification and consistency. To streamline the testing process, an Android app was developed, enabling the selection of the reasoner, ontology set (based on OWL profile sublanguages), and inference task, and storing results in an embedded database.

Evaluation frameworks specialised for non-standard inference test cases also exist. *JustBench* [7] analyses reasoner performance on testing the correctness of *justifications*, *i.e.*, minimal ontology subsets for an entailment to hold. Experiments concerning the *Mini-ME* (Mini Matchmaking Engine) Java/Android reasoner [107] and its *Swift* reengineering for iOS [98] have evaluated performance of the *matchmaking* task. More recently, the interest in applying semantic technologies to ubiquitous computing has generated the need for benchmarking mobile-oriented OWL profiles and emerging mobile reasoners. The experimental campaign in [18] evaluated six reasoners with *OWL API* [49] support on Android, using the ORE 2013 dataset [39], on ontology classification and consistency tasks. In order to automate the large number of tests, an Android application was developed, which allowed selecting the reasoner, the set of ontologies (based on an OWL profile sublanguage) and the inference task, and then saved results in an embedded database.

Platform heterogeneity and strict energy usage control are among the distinctive traits of mobile and ubiquitous computing, therefore cross-platform and energy-aware benchmarking frameworks are currently at the edge of research and development efforts. The framework in [129], aimed at evaluating mobile semantic rule engines, has been developed in JavaScript exploiting the *PhoneGap*[3] Software Development Kit (SDK): this approach allowed harnessing rule engines written either in JavaScript or natively for one of the platforms supported by PhoneGap (Android, iOS, Windows 8.1). A recent and enhanced version of the framework, named *MobiBench* [128], additionally supports OWL 2 RL reasoning, benchmark automation, and Java reasoners via the *Nashorn* JavaScript engine included in Java SE version 8 and later. Energy usage profiling is planned for future work.

While there are some energy-aware mobile benchmarks, they usually serve specific, one-time research efforts. The study in [86] assessed Android reasoners by replacing the testbed device battery with a hardware power monitor for precise data capture. Despite its accuracy, this method is complex due to varying electrical parameters across mobile models; the current prevalence of non-removable batteries in smartphone models further complicates this approach. Alternatively, [61] used an *ODROID XU3* single-board computer with integrated power monitoring circuitry to evaluate six reasoners. This method proved to be more practical than [86], but the chosen hardware does not fully represent typical mobile and ubiquitous computing environments, and the study limited its analysis to Java-based reasoners. A more software-centric approach was adopted in [31], where a profiler correlated battery charge and time metrics using Android APIs, achieving accuracy within 5% of hardware monitors for apps with minimal network or sensor usage. This method was also employed in [125] for benchmarking Android reasoners' energy consumption. In [43], a similar tool was used to develop a model predicting energy usage for mobile ontology reasoning. Key findings include: (i) energy consumption is influenced by the battery's charge state even for the same device-reasoning task pair, and (ii) the correlation between task duration

[3]PhoneGap home: `https://phonegap.com/`. It is based on the open source *Apache Cordova* engine: `https://cordova.apache.org/`

and power consumption is not always linear, meaning longer tasks are not necessarily more energy-intensive. These results call for further investigation, and underline the need for a scalable evaluation framework that efficiently adapts to various mobile devices and platforms, offering broader insights into energy usage in mobile reasoning tasks.

4.2 Using evOWLuator

This section details how evOWLuator can be configured and used to run evaluations and visualize results. Installation instructions and further technical details, including command-line options and flags, are provided in the online documentation.[4] Once the tool is installed and correctly configured, it can be used by invoking the `evowluate` command line tool, followed by a *subcommand* representing the specific task that should be carried out by the framework, as pictured in Figure 4.1. Available subcommands are introduced in what follows.

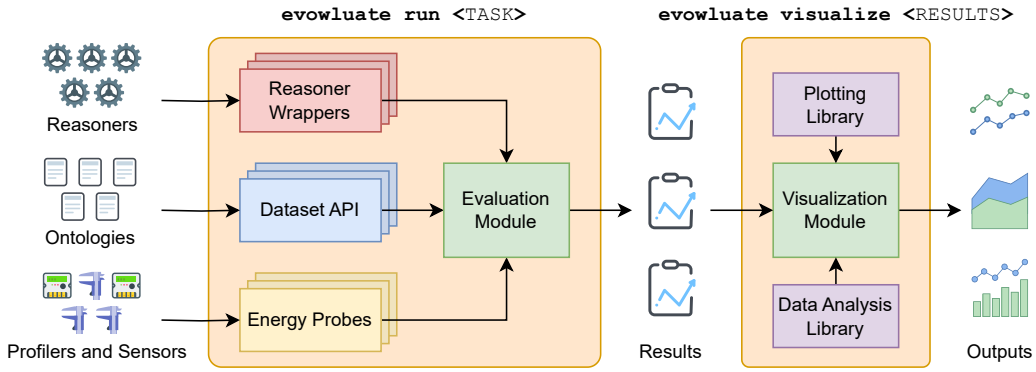


Figure 4.1: evOWLuator high-level architecture and data flow.

[4] evOWLuator documentation: `http://swot.sisinflab.poliba.it/evowluator`

4.2.1 Setup

Once the framework has been installed, running an evaluation requires some preliminary setup:

- **Datasets:** ontology corpuses must be placed in the `data` directory.[5] Each dataset must have a root folder, whose name is used by EVOWLUATOR as the dataset name, and a subfolder for each supported syntax,[6] which in turn must contain ontologies serialized in the specified syntax.
- **Reasoners, reasoning tasks, energy probes:** must be configured by placing Python modules implementing the `Reasoner`, `ReasoningTask` and `EnergyProbe` interface into the `evowluator/user/reasoners`, `evowluator/user/tasks`, and `evowluator/user/probes` directories, respectively.

Missing dataset serializations can be automatically produced by EVOWLUATOR through the `convert` subcommand, which allows translating datasets into any of the formats supported by the framework. Further details about the plugin mechanism for reasoners, tasks, and energy probes are provided in Section 4.4.

4.2.2 Running evaluations

After setting up all necessary components, evaluations can be started through the `run` subcommand, followed by the reasoning task to evaluate and other mandatory arguments, such as the dataset to use for the evaluation. EVOWLUATOR provides built-in support for some standard (*classification* and *consistency*) and non-standard (*matchmaking*) tasks. Custom reasoning tasks

[5]References to filesystem paths are relative to the EVOWLUATOR root directory.

[6]Supported syntaxes: `dl`, `functional`, `krss`, `krss2`, `manchester`, `obo`, `owlxml`, `rdfxml`, `turtle`.

can be easily added by implementing the `ReasoningTask` interface, detailed in Section 4.4.2. Each inference task can be evaluated in two modes:

- **Correctness**: checks the validity of inference outcomes. It is possible to use a single reasoner as a correctness oracle, or to evaluate correctness via consensus through a randomized majority vote. A further correctness strategy involves assuming all reasoners return correct results, and only accounting for runtime errors or timeouts. In any case, under this evaluation mode, reasoner outputs are collected rather than just correctness results, therefore the desired correctness strategy can be changed *a posteriori* when visualizing results. For reasoning tasks that return sizable outputs, such as ontology classification, a hash of the output is stored.
- **Performance**: collects inference performance statistics, in terms of time and maximum memory usage, and optionally about energy usage, provided that the user specifies one or more *energy probes* via command-line flags. The framework is able to collect and visualize multiple time measurements from each reasoner, which are interpreted as separate reasoning phases (*e.g.*, parsing, preprocessing, reasoning, etc.).

Test execution can be controlled through a number of flags, allowing the user to control various aspects of the evaluation:

- By default, evaluations are run for all reasoners that support the specified reasoning task, though they can be restricted to specific reasoners.
- It is possible to specify a timeout for inferences, after which the reasoner process is killed.
- Performance tests can be run for multiple iterations, which are averaged by the framework when producing visualizations.
- Correctness tests can be parallelized by specifying a certain number of worker processes.

Evaluations can be stopped by sending the `SIGINT` POSIX signal to the framework, *e.g.*, by pressing `CTRL+C` in the shell. Interrupted or otherwise incomplete evaluations can be resumed through the `resume` subcommand. Once a test is completed, the framework outputs and stores the following items in a new subdirectory within the `results` dir:

- a human-readable *log* of the assessment;
- a summary of the *configuration* used for the evaluation (selected reasoners, dataset, syntaxes, *etc.*);
- machine-processable test *outcomes*, whose content and structure vary depending on the configuration.

4.2.3 Visualizing results

Other than producing raw evaluation results, evOWLuator can generate aggregate reports and graphical plots. Available visualization types vary depending on the test configuration:

- For **correctness** tests, the tool outputs statistics about the number of correct and incorrect results, runtime errors and timeouts, and displays a grouped bar plot depicting these metrics (*e.g.*, Figure 4.6).
- For **performance** tests, the framework computes per-ontology and dataset-wide times, as well as information about the minimum, maximum, and average detected memory peak, for each reasoner. If energy probes are specified, then aggregate metrics are provided for energy consumption as well. Produced plots are grouped bar charts for cumulative results (*e.g.*, Figure 4.7a, Figure 4.7c) and scatterplots of the evaluated metric by ontology size (*e.g.*, Figure 4.7b, Figure 4.7d).

Plots are displayed in an interactive window, which can be used for navigation, zooming and cropping, though they can also be saved as vector or

raster graphics files. Multiple aspects of the plots can be configured through dedicated command-line arguments, such as:

- plot size, titles, and labels;
- axis scale and limits;
- time and memory units;
- legend location and layout;
- types and colors for markers and bars;
- polyline fit for scatterplots, and degree of the fitted polynomial.

4.3 Architecture

evOWLuator's design focuses on flexibility, especially in terms of its capability to test various reasoning engines and to run inference services on mobile and embedded devices. To meet these objectives, evOWLuator employs an *object-oriented* approach: users configure the framework by extending Python's abstract base classes with concrete subclasses, which implement the interfaces of their parents. While this *programmatic* method might be more verbose compared to a *declarative* approach, like using structured configuration files, it offers greater expressiveness, as it allows users to leverage the full capabilities of the Python programming language and its standard library. This design choice significantly enhances the framework's adaptability and future-proofing, making it flexible enough to allow the integration of arbitrary reasoner interfaces.

The key components of the framework are detailed below and illustrated in the Unified Modeling Language (UML) component diagram in Figure 4.2. The *Data* module provides the `Dataset` and `Ontology` classes, facilitating access to datasets provided by the user and the ontologies they contain. The

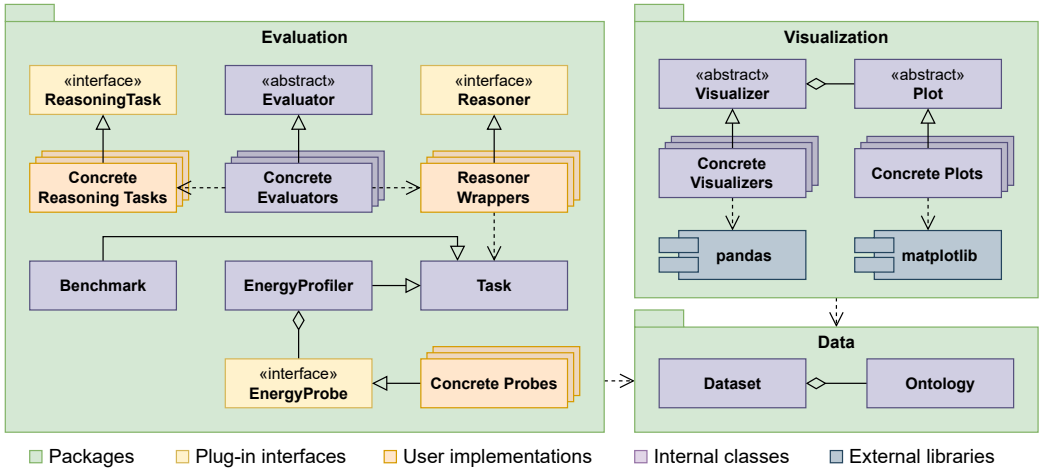


Figure 4.2: UML diagram of the main components of EVOWLUATOR.

Evaluation engine interfaces with reasoners, ontologies, and energy profilers through customizable software endpoints. Most of the functionality is encapsulated within *Evaluators*, subclasses of the `Evaluator` abstract class implementing core business logic for each type of evaluation, such as correctness or performance. Inference task details are modeled by `ReasoningTask` and implemented by concrete classes adhering to this interface. `Task` is the core API for spawning processes and capturing their output: its subclasses allow profiling energy consumption and benchmarking execution times and memory usage.

The *Evaluation Engine* component invokes inference tasks implemented by reasoners through user-provided subclasses of the `Reasoner` abstract class, which supports running tests on the local machine as well as orchestrating them on remote devices. The latter option is particularly aimed at mobile and embedded devices, with out-of-the-box support included for both iOS and Android platforms (refer to Section 4.4.1). For performance and energy footprint evaluations, the engine can execute multiple test iterations as specified by command line arguments. When more than one iteration is executed, the framework averages the results to provide a consolidated output for further processing. Additionally, users can set a *timeout* for each reasoning task, after which the reasoner process is automatically terminated. EVOWLUATOR is equipped to detect *runtime errors* in two scenarios: the

reasoner exits with a non-zero code, or it fails to produce essential output, like computation times for performance evaluations or inference results for correctness checks. Upon completion of a test, the *Evaluation Engine* records the outcomes, which can then be used for visualization. The *Visualization Engine* is able to create tabular summaries and graphical representations, providing human-understandable recaps of raw results. This component comprises subclasses of the `Visualizer` abstract class, with charting and plotting functionalities offered by subclasses of the `Plot` abstract class.

4.4 Available interfaces

4.4.1 Reasoners

To be integrated with evOWLuator for local invocation, reasoners need a command line interface capable of executing reasoning tasks on specific ontologies. At a minimum, they should accept inputs specifying the reasoning task and the path to an ontology file. The details of the arguments vary for each reasoner and are defined through the implementation of the `Reasoner` interface. All subclasses of the `Reasoner` abstract class are dynamically loaded by the framework. Each subclass must include metadata about the encapsulated reasoner, such as its name, executable file path, supported OWL syntaxes, and inference services. Additionally, the subclass is required to define the command line argument array for each reasoning task it supports. Regarding correctness results, command line tools should either conform to the output format expected by evOWLuator or, alternatively, custom output parsing logic can be implemented by creating a subclass of the `ResultsParser` class. This flexibility ensures that the framework can accommodate a wide range of reasoners and their respective output formats.

In addition to the `Reasoner` base class, the proposed framework provides a few templates to simplify the integration of inference engines on notable platforms, pictured in Figure 4.3 and described hereafter.

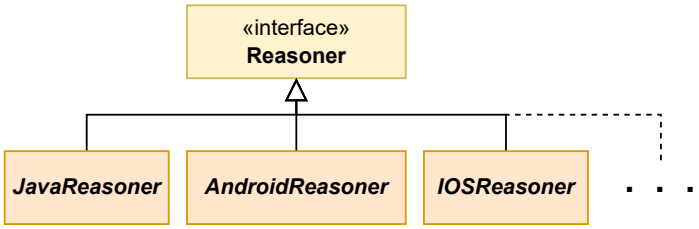


Figure 4.3: *Reasoner* interface and template classes.

- **Java SE:** this template facilitates the integration of Java reasoning engines compiled in *jar* files by abstracting away the instantiation of the *Java Virtual Machine* (JVM). JVM configuration is controlled through appropriate flags specified with dedicated methods.

- **iOS:** the template enables running and testing iOS-based reasoners. In this case, inference engines have to be wrapped in *Xcode* projects, and specifically as Xcode test cases, *i.e.*, `XCTestCase`[7] subclasses. Supported reasoning tasks are exposed by means of dedicated test case methods to be deployed to the target device, together with datasets for the evaluation. evOWLuator invokes test cases through `xcodebuild`, Xcode's command line interface, passing any required data via environment variables. In this case, in the user-provided extension of the template, methods just need to return project-related information, such as the path to the Xcode project and the name of the test methods implementing each of the supported reasoning tasks.

- **Android:** the template allows the framework to run and test reasoning tasks on Android devices. Similarly to iOS, user-implemented methods just need to return Android-specific information, such as the package identifier of the reasoner app. Users must install reasoners as Android wrapper applications containing an `EVOWLUATE` intent filter. evOWLuator automatically installs a *launcher* application used to start reasoner apps by issuing appropriate `EVOWLUATE` intents,[8] and to close them once the reasoning task is over. The implementation

[7]XCTest: `https://developer.apple.com/documentation/xctest`
[8]Android intents: `https://developer.android.com/reference/android/content/Intent`

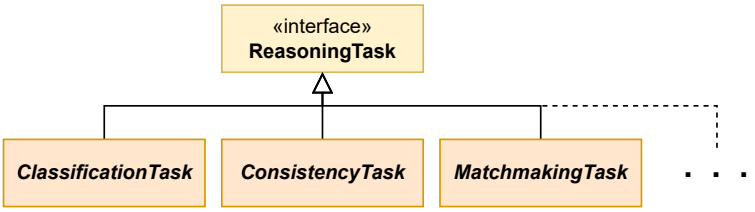


Figure 4.4: *ReasoningTask* interface and built-in tasks.

of this component exploits the Android `Instrumentation` class.[9] All communications between evOWLuator and the launcher application are carried out through the *Android Debug Bridge* (`adb`), which must be installed on the host machine. *USB debugging* must also be enabled via the Settings app of the target mobile device, by accessing the hidden *developer menu*.[10]

4.4.2 Reasoning tasks

Details that are specific for each reasoning task are modeled by the `ReasoningTask` interface, which allows the specification of expected inputs and outputs for each inference service. As an example, while ontology classification and consistency only require one input file (the ontology to process), other inferences may require additional inputs (*e.g.*, matchmaking also requires an OWL ontology containing individuals acting as requests). Expected reasoner outputs are also controlled by this interface, as in what follows.

- For **correctness**, results may be returned on the standard output, as text files, or as ontology files. In the latter case, the output must be a valid OWL ontology in any of the serializations supported by the OWL API [49]. As an example, reasoners must return results on the standard output for the consistency task, while they are expected to return the inferred taxonomy as an OWL ontology when computing classification.

[9]Android Instrumentation: `https://developer.android.com/reference/android/app/Instrumentation`

[10]Android developer menu: `https://developer.android.com/studio/debug/dev-options`

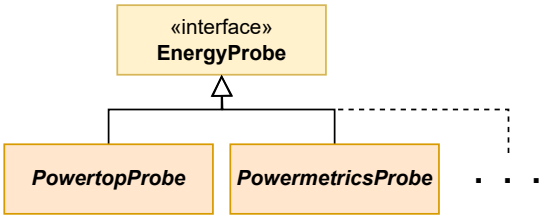


Figure 4.5: *EnergyProbe* interface and built-in probes.

- With respect to **performance**, the interface specifies which results the reasoners are expected to return on the standard output, *i.e.*, reasoning phases with corresponding time measurements. It is also possible to specify whether reasoners must provide a memory reading, which is necessary for reasoning tasks deployed to remote devices. For local reasoners, the framework defaults to computing the memory metric as the *maximum resident set size* (MRSS) of the reasoner process, measured by means of the cross-platform `psutil`[11] Python library. Remote reasoners must compute this metric themselves, usually in platform-specific ways: as an example, iOS and Android reasoners can do so through the `getrusage`[12] POSIX call.

4.4.3 Energy footprint

Energy drain estimation is implemented by the `EnergyProfiler` class, which runs the reasoner and polls a user-specified *energy probe* instance while the related process is alive. Energy probes must implement the `EnergyProbe` interface, as shown in Figure 4.5. The N collected samples are then used to compute an *energy footprint* score, providing an estimation of the energy employed by the engine during its execution:

$$score = sampling\_interval * \sum_{i=1}^{N} sample_i \tag{4.1}$$

[11] Process and system utilities (psutil): `https://psutil.readthedocs.io`
[12] `getrusage` man page: `https://linux.die.net/man/2/getrusage`

The approach is designed for broad compatibility with both software- and hardware-based energy metering solutions. `EnergyProbe` instances can interface with built-in power management tools of the operating system, utilize data from energy profilers (such as those outlined in Section 4.1), or even integrate readings from external hardware devices, such as the Monsoon High Voltage Power Monitor.[13] EvOWLuator natively supports the *PowerMetrics*[14] (developed by Apple Inc. for macOS) and *PowerTOP*[15] (created by Intel Corp. for GNU/Linux and also functional on Microsoft Windows via WSL) software-based energy profilers. To incorporate additional energy probes, users can create classes that implement the `EnergyProbe` interface, which must compute and record power usage samples, reflecting the average energy consumption over the interval between consecutive polls.

Energy consumption is returned as an absolute *score* without a specific measurement unit, a necessary adaptation for compatibility with certain energy profilers like PowerMetrics, which do not provide power usage in standard physical units. Nonetheless, following Equation 4.1, when power samples are measured in watts, the score can indeed be interpreted as energy consumption in joules. This interpretation is applicable to tools like PowerTOP and potentially to hardware-based power meters.

When testing on battery-equipped devices, consistency in power source (grid or battery) and charge level at the start of each test is crucial. Variations in these factors can affect energy readings, thereby impacting test reproducibility. For cross-device or cross-operating-system energy comparisons, the framework should ideally be used in conjunction with an external hardware power meter. This method provides the most accurate, device-independent energy measurements; however, it is important to cautiously analyze the results, considering potential biases or confounding factors due to differences in the hardware and software of the devices under test.

[13] Monsoon HV Monitor: `https://www.msoon.com/high-voltage-power-monitor`

[14] PowerMetrics: `https://developer.apple.com/library/archive/documentation/Performance/Conceptual/power_efficiency_guidelines_osx`

[15] PowerTOP: `https://www.intel.com/content/www/us/en/developer/articles/tool/powertop-primer.html`

4.5 Case study: benchmarking classification and consistency

To validate evOWLuator's functionality and illustrate its application, a demonstration experimental campaign has been conducted, focusing on the correctness, performance, and energy footprints of ontology classification and ontology consistency inference services offered by selected state-of-the-art OWL reasoners, with tests performed on both desktop and mobile devices. As the primary aim is to test the framework's capabilities as a benchmarking platform, the scope of this campaign has been intentionally limited, involving a smaller subset of reasoners and datasets compared to larger-scale tests referenced in Section 4.1. This approach allows for a focused assessment of evOWLuator's key features and effectiveness in a controlled setting.

4.5.1 Testbed, reasoners and datasets

Desktop tests have been performed on a *Linux* workstation[16] and a *Mac Mini (2014)*,[17] while mobile experiments have been carried out on an *Apple iPhone 7*[18] and a *HTC/Google Nexus 9* tablet.[19] Tested desktop reasoners include: *Fact++* (version 1.6.5) [124], *HermiT* (1.3.8) [38], *Konclude* (0.6.2-544) [115], *Mini-ME* (2.0) [107], *Mini-ME Swift* (1.0) [98] and *TrOWL* (1.5) [123]. Mini-ME and Mini-ME Swift have also been used for tests on Android and iOS, respectively. Additionally, the *JFact* (1.2.1), *HermiT* (1.3.8) and *Pellet* (2.3.1) [113] Android ports from [18] have been evaluated. Tests have been carried out on the following datasets:

[16]AMD Ryzen 5 3600 CPU at 3.6 GHz, 32 GB DDR4 RAM at 3000 MT/s, 1 TB SSD, Ubuntu 18.04 LTS x64.

[17]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.

[18]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

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

- **ORE 2014**: 1398 ontologies selected from the *2014 OWL Reasoner Evaluation Workshop* competition dataset,[20] considering only those having at most $\mathcal{ALN}$ as reference expressiveness. This allows demonstrating the evaluation of Mini-ME, which does not support more expressive languages.

- **ORE 2014 Energy**: it consists of the 50 largest ontologies from the ORE 2014 set (average size 1781.28 ± 14063.8 KiB, minimum 11.44 KiB, maximum 291.33 MiB in functional syntax), and it was specifically used for conducting energy consumption tests on macOS. The focus on large ontologies stems from the need for sufficiently long-running processes to ensure accurate readings from software energy profilers like *PowerMetrics* and *PowerTOP*. These tools often fail to generate data for short-lived processes which start and end before any power sample is taken, even if such processes are highly energy-intensive. This selective approach addresses the limitations of the profilers, ensuring reliable energy consumption data for longer-duration tasks.

- **BioPortal**: a dataset composed of 20 ontologies from BioPortal[21] [135] with no restrictions on DL expressiveness (average size 130.79 ± 104.57 MiB, minimum 27.92 MiB, maximum 450.71 MiB in functional syntax). BioPortal has been chosen because life sciences ontologies are among the largest as well as the best known by practitioners of Semantic Web technologies. Ontologies have been selected by taking the 100 largest ones in BioPortal, classifying them via four high-expressiveness reasoners (Konclude, Fact++, HermiT and TrOWL), and selecting only those for which all reasoners returned correct and complete inferences within 2 hours. This dataset has been used to demonstrate energy profiling capabilities on both Linux and macOS.

[20] `http://dl.kr.org/ore2014`

[21] `https://bioportal.bioontology.org`

4.5.2 Setup

Experiments setup has followed the procedure reported in detail in EVOWLUATOR's documentation.

Datasets

After installing EVOWLUATOR on both desktop machines in an `evowluator` base directory, the aforementioned datasets have been set up by moving ontologies into appropriate subdirectories of the `data` directory in the install path. Each dataset is expected to have a root directory, whose name is used as identifier, and a subdirectory for each supported OWL syntax, which in turn must contain ontology files. Missing ontology formats, as needed by some of the reasoners, have been generated using the `convert` subcommand.[22]

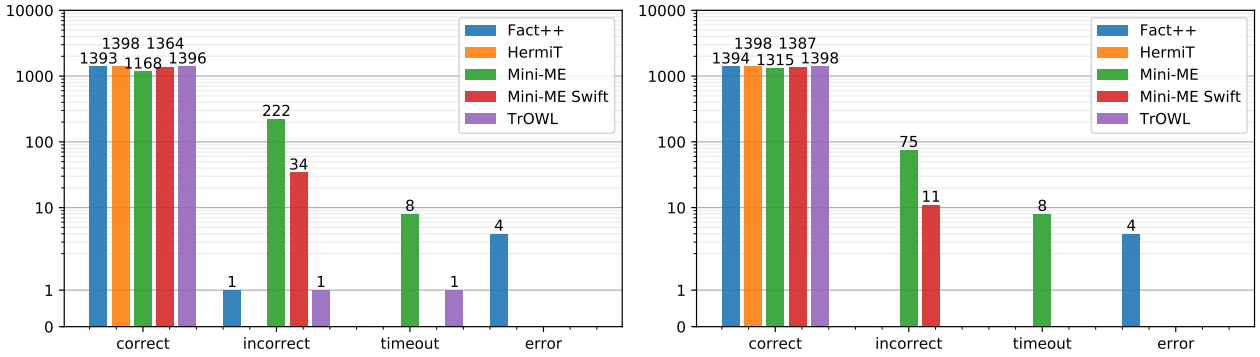
Desktop reasoners

The next step has concerned the integration of reasoners, which (as recalled in Section 4.4.1) must be configured by writing Python modules implementing the `Reasoner` interface. For Java-based reasoners supporting the OWL API, a wrapper has been created to expose the classification and consistency inference tasks, ensuring all reasoners share the same command line interface. This has covered Fact++, HermiT, TrOWL and Mini-ME. The Python side of the integration has then been accomplished by writing a single `Reasoner` template subclass for all OWL API reasoners, further subclassed to configure individual reasoner metadata. Mini-ME Swift and Konclude come with a built-in command line interface, instead, so providing befitting `Reasoner` subclasses has been enough to integrate them.

iOS reasoners

After installing Xcode and its command line tools on the host machine, supporting iOS reasoners like Mini-ME Swift has required creating a `XCTest`

[22]The ORE 2014 dataset is only available in functional OWL syntax, though Mini-ME Swift requires ontologies in RDF/XML format.



(a) Classification. **(b)** Consistency.

Figure 4.6: Correctness results of various OWL reasoners.

project and wrapping reasoning task invocation in separate methods of an `XCTestCase` subclass. With respect to the Python side of the integration, the `IOSReasoner` template class has been subclassed, specifying appropriate metadata. Ontologies have been uploaded to the target device through the *copy bundle resources* Xcode build phase.

Android reasoners

The `adb` tool is required on the host computer to enable communication with the target Android device. Straightforward wrapper Android apps have been implemented for Mini-ME, JFact, HermiT and Pellet, which respond to specific intents and start the corresponding reasoning tasks. The Python side of the configuration has involved subclassing the `AndroidReasoner` template provided by evOWLuator. Ontologies have been uploaded to the storage memory of the device prior to starting the tests.

4.5.3 Results

Correctness

Correctness has been checked on the whole ORE 2014 dataset using Konclude as test oracle, since the latest OWL reasoner competition [84] reported it as the most reliable with regard to ontology classification and consistency. The

Table 4.1: Summary of classification correctness tests.

Reasoner	Correct	Incorrect	Timeout (s)	Error	Ratio
Fact++	1393	1	0	4	1.00
HermiT	1398	0	0	0	1.00
Mini-ME	1168	222	8	0	0.84
Mini-ME Swift	1364	34	0	0	0.98
TrOWL	1396	1	1	0	1.00

Table 4.2: Summary of consistency correctness tests.

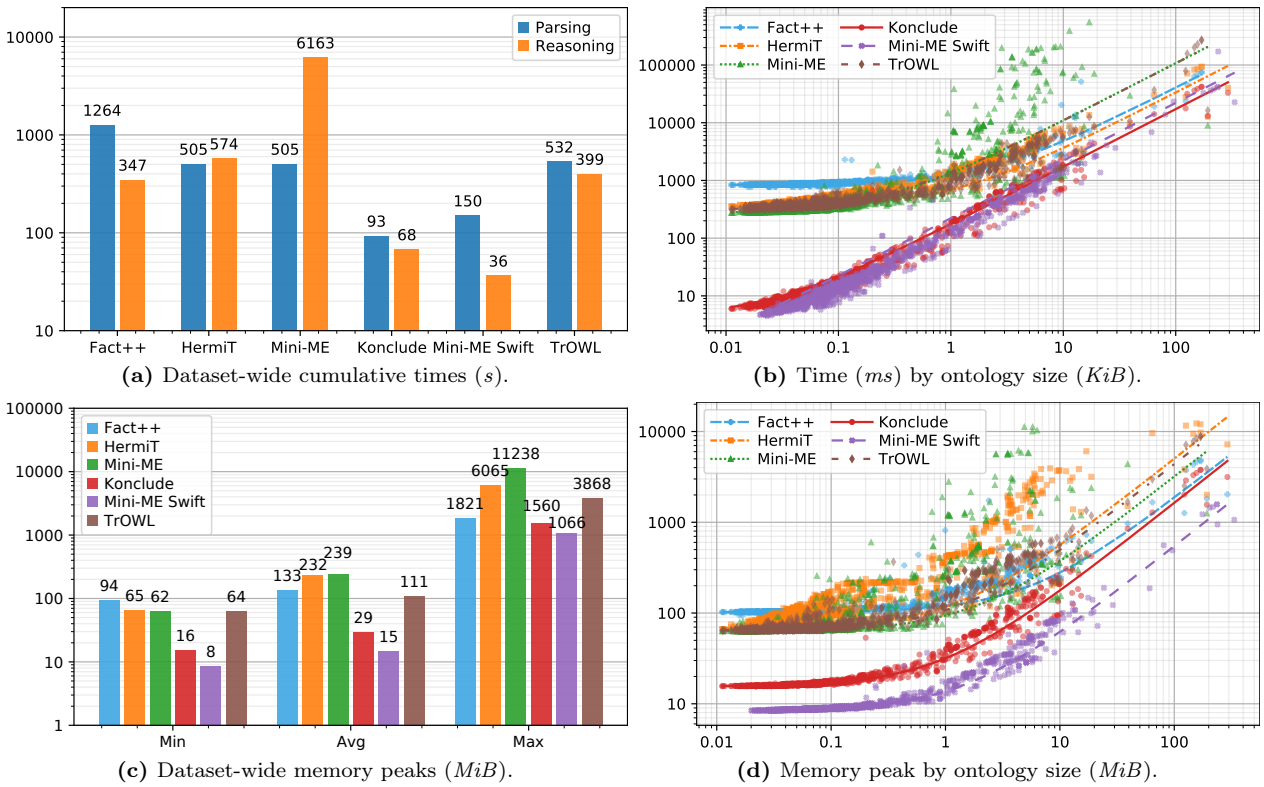
Reasoner	Correct	Incorrect	Timeout (s)	Error	Ratio
Fact++	1394	0	0	4	1.00
HermiT	1398	0	0	0	1.00
Mini-ME	1315	75	8	0	0.94
Mini-ME Swift	1387	11	0	0	0.99
TrOWL	1398	0	0	0	1.00

outcomes are illustrated in Figure 4.6, showing the number of correct and incorrect results for each reasoner with respect to the test oracle (which does not appear in the plot, of course). The plot also displays the number of times reasoners have hit the imposed timeout, and the number of runtime errors. The same data is also reported in Tables 4.1 and 4.2, which further include a correctness ratio, computed as the number of correct results over the number of ontologies in the dataset. It can be noted Mini-ME and Mini-ME Swift exhibit lower ratios than the other reasoners: incorrect results for Mini-ME and Mini-ME Swift are due to unsupported constructs in the ontologies; the former has additional timeouts on the largest ontologies of the dataset.

Desktop performance

Performance evaluation metrics refer to the ontologies from the ORE 2014 dataset which all the above reasoners have classified correctly within the timeout on the macOS testbed. Results for classification and consistency are pictured in Figures 4.7 and 4.8, respectively, evidencing evOWLuator's capability to generate histograms and scatterplots. In detail:

- Figures 4.7a and 4.8a show histogram plots of dataset-wide cumulative parsing and reasoning times in seconds;



(a) Dataset-wide cumulative times (s).

(b) Time (ms) by ontology size (KiB).

(c) Dataset-wide memory peaks (MiB).

(d) Memory peak by ontology size (MiB).

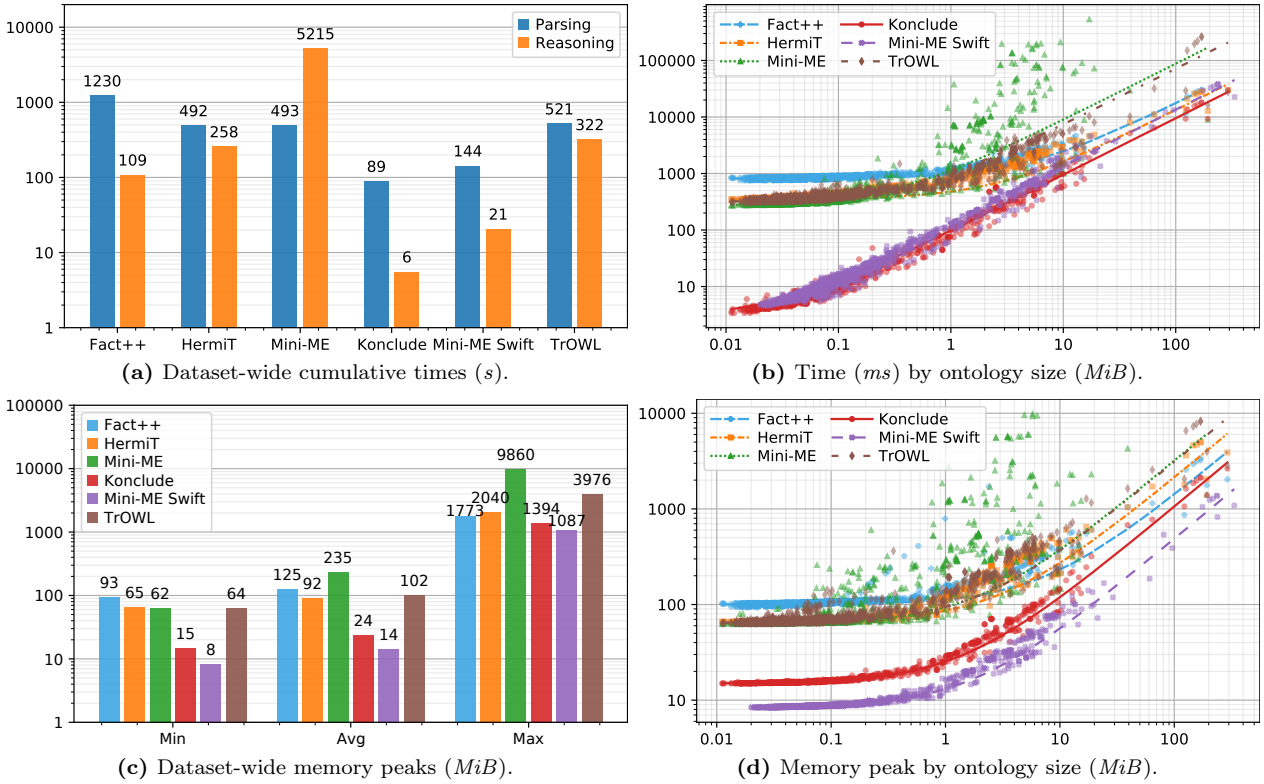
Figure 4.7: Classification performance tests on desktop.

- Figures 4.7b and 4.8b depict time as a function of ontology size;
- Figures 4.7c and 4.8c illustrate dataset-wide minimum, average and maximum memory peaks for each reasoner;
- Figures 4.7d and 4.8d plot memory peak as a function of ontology size.

Aggregated results output by the framework are summarized in Tables 4.3 and 4.4, showing total parsing and reasoning time, as well as dataset-wide minimum, average and maximum memory peak for each reasoner.

Mobile performance

Similar plots are shown for mobile tests, sketching performance metrics of Android reasoners (Figures 4.9 and 4.10) and Mini-ME Swift running on iOS



(a) Dataset-wide cumulative times (s).

(b) Time (ms) by ontology size (MiB).

(c) Dataset-wide memory peaks (MiB).

(d) Memory peak by ontology size (MiB).

Figure 4.8: Consistency performance tests on desktop.

(Figures 4.11 and 4.12). Both desktop and mobile outcomes are in line with previous experimental campaigns [98].

It can be noticed how EVOWLUATOR uses *Matplotlib* to automatically adapt and differentiate graph elements (size and color) of reports depending on the number of tested reasoners. All figures in this section have been generated in Portable Document Format (PDF) and integrated with no modification in the LaTeX project of this dissertation; likewise they can be formatted to Scalable Vector Graphics (SVG) or Portable Network Graphics (PNG) for Web publishing.

ORE 2014 Energy footprint

These tests measure the energy consumption of reasoning tasks on the ORE 2014 Energy dataset. All tests have been executed on the macOS testbed.

Table 4.3: Summary of classification performance tests on desktop.

Reasoner	Parsing time (s)	Reasoning time (s)	Total time (s)	Min mem. peak (MiB)	Avg mem. peak (MiB)	Max mem. peak (MiB)
Fact++	1247.31	347.48	1594.79	93.57	131.00	1728.90
HermiT	491.35	573.57	1064.92	65.45	227.37	6064.55
Konclude	76.28	67.64	143.91	15.53	27.39	455.15
Mini-ME	491.44	6162.98	6654.41	62.49	233.34	11237.72
Mini-ME Swift	121.37	36.40	157.77	8.41	13.91	189.50
TrOWL	516.93	399.39	916.32	63.88	105.78	1353.89

Table 4.4: Summary of consistency performance tests on desktop.

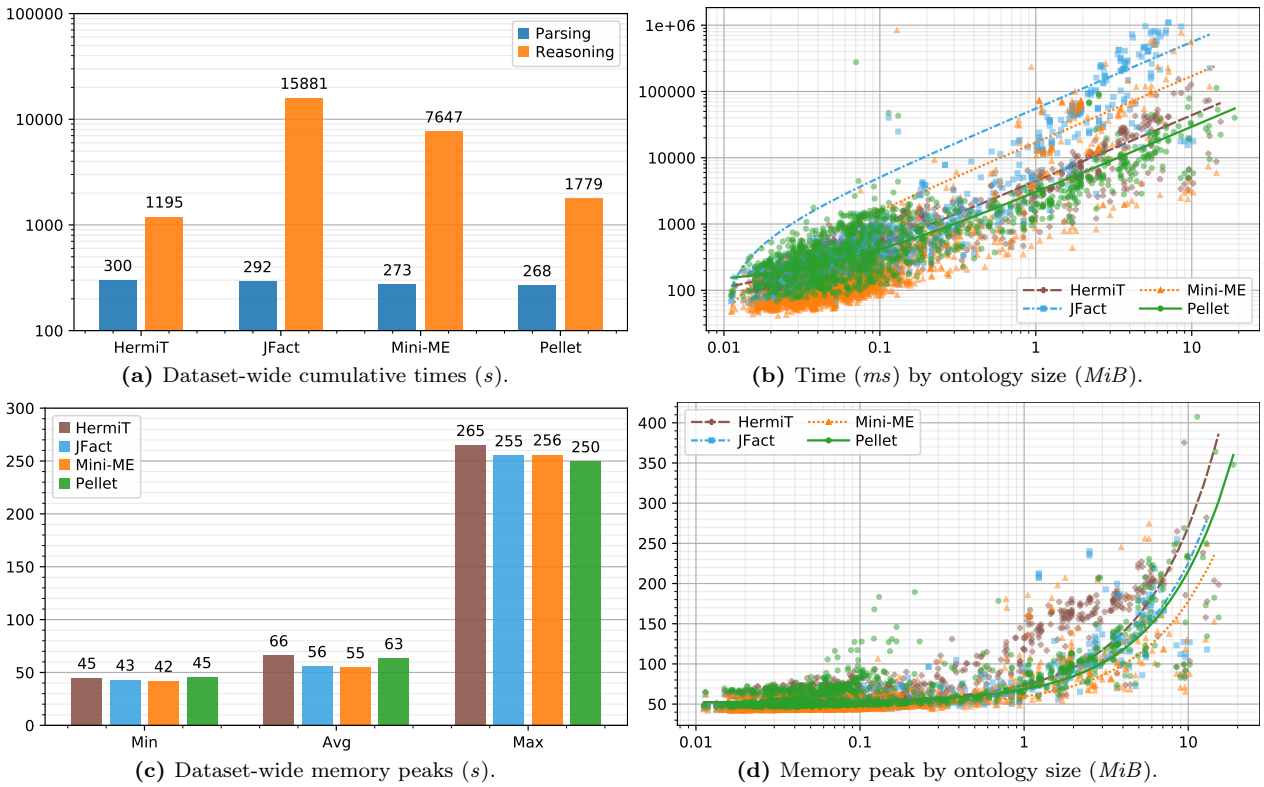
Reasoner	Parsing time (s)	Reasoning time (s)	Total time (s)	Min mem. peak (MiB)	Avg mem. peak (MiB)	Max mem. peak (MiB)
Fact++	1212.18	108.55	1320.73	92.98	122.94	1718.54
HermiT	477.92	258.24	736.16	64.95	89.34	649.09
Konclude	72.57	5.58	78.15	14.96	22.19	245.55
Mini-ME	479.05	5214.83	5693.88	62.38	229.72	9859.68
Mini-ME Swift	114.93	20.69	135.62	8.36	13.16	167.49
TrOWL	506.20	321.51	827.71	63.71	97.32	1278.56

Results are shown in Figure 4.13; it is important to recall they represent energy usage, therefore lower scores are better. In particular:

- Figures 4.13a and 4.13c recall dataset-wide minimum, average and maximum energy footprint for each reasoner;
- Figures 4.13b and 4.13d plot energy footprint as a function of the ontology size;
- Tables 4.5 and 4.6 summarize energy evaluation results: they provide the same information as Figures 4.13a and 4.13b, though in tabular form.

It should be noticed how Fact++ and HermiT have a significantly smaller energy footprint for consistency than classification, while for the other reasoners the two scores are closer.

The availability of results in CSV format facilitates further processing through the *pandas* Python library. As an example, Table 4.7 has been created



(a) Dataset-wide cumulative times (s).

(b) Time (ms) by ontology size (MiB).

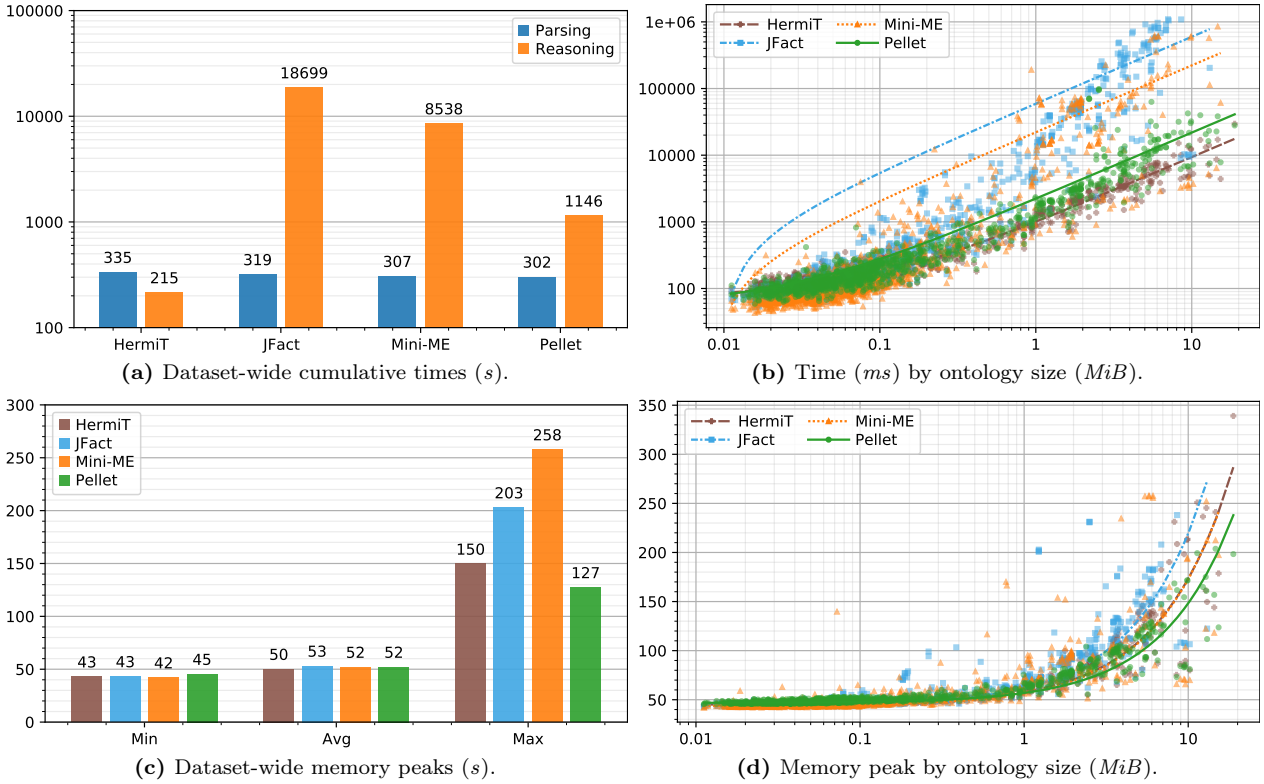
(c) Dataset-wide memory peaks (s).

(d) Memory peak by ontology size (MiB).

Figure 4.9: Classification performance tests on Android.

by computing the time-energy, memory-energy and time-memory Pearson correlation coefficients for each reasoner for the classification task, derived on the ORE 2014 Energy dataset. They have been computed as follows: average results output by performance and energy evaluations are loaded and merged through the `DataFrame.merge()` method; pairwise correlation between columns is then computed via the `DataFrame.corr()` method.[23] Values indicate a strong linear correlation between energy footprint and time for all reasoners, in accordance with [86] but partial disagreement with [43], as discussed in Section 4.1. Linear correlation has also been found between energy and memory usage in the majority of tested engines, with the exceptions of Fact++ and Mini-ME, whose behavior calls for further investigation through specific experiments.

[23]Pandas DataFrame documentation: `https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html`



(a) Dataset-wide cumulative times (s).

(b) Time (ms) by ontology size (MiB).

(c) Dataset-wide memory peaks (s).

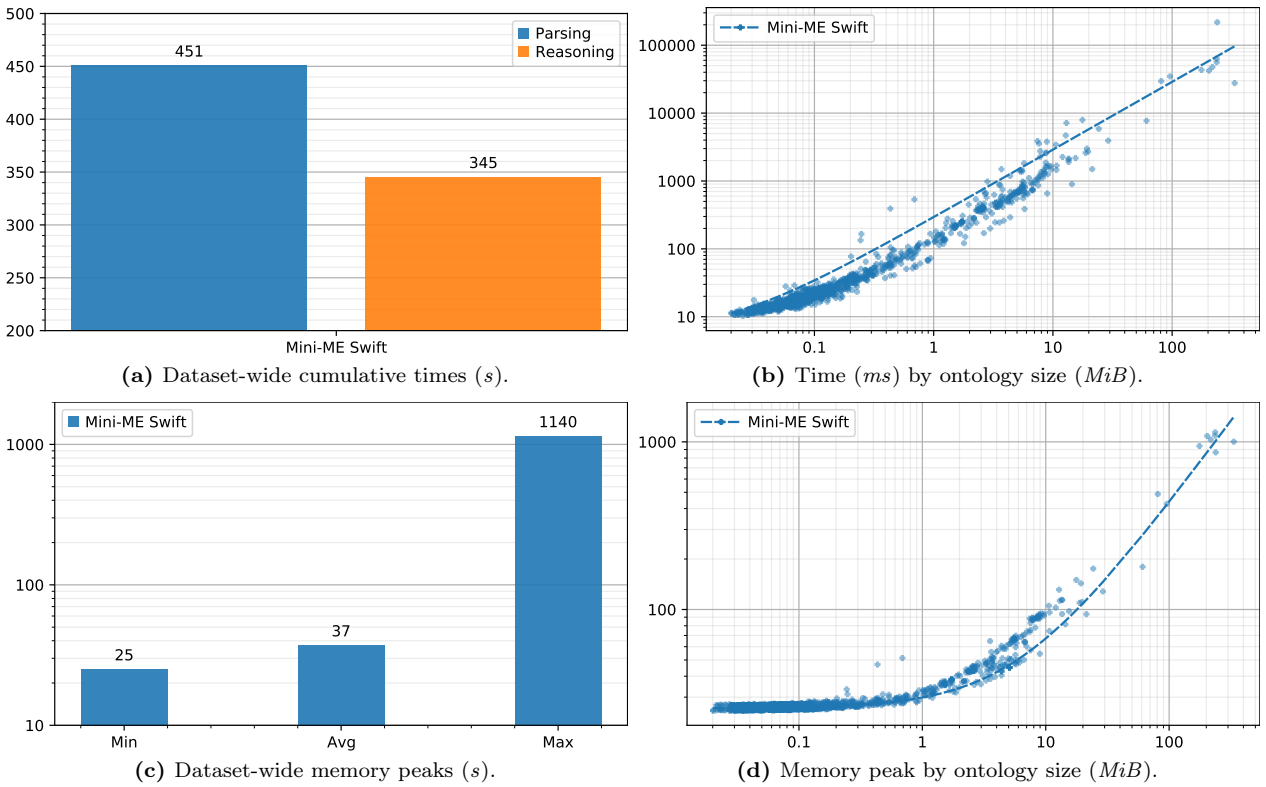
(d) Memory peak by ontology size (MiB).

Figure 4.10: Consistency performance tests on Android.

BioPortal Energy Footprint

In order to assess EVOWLUATOR features on multiple platforms and on very large and expressive ontologies, an additional experimental session has been carried out, measuring energy footprint on both Linux and macOS on the BioPortal Energy dataset. Mini-ME Swift has been excluded from this test as its supported expressiveness is limited to $\mathcal{ALN}$. Results are summarized as follows:

- Figure 4.14 reports on energy scores measured on Linux via `powertop`;
- Tables 4.8 and 4.9 refer to Linux tests in a tabular form;
- Figure 4.15 displays energy scores measured on macOS through `powermetrics`;



(a) Dataset-wide cumulative times (s).

(b) Time (ms) by ontology size (MiB).

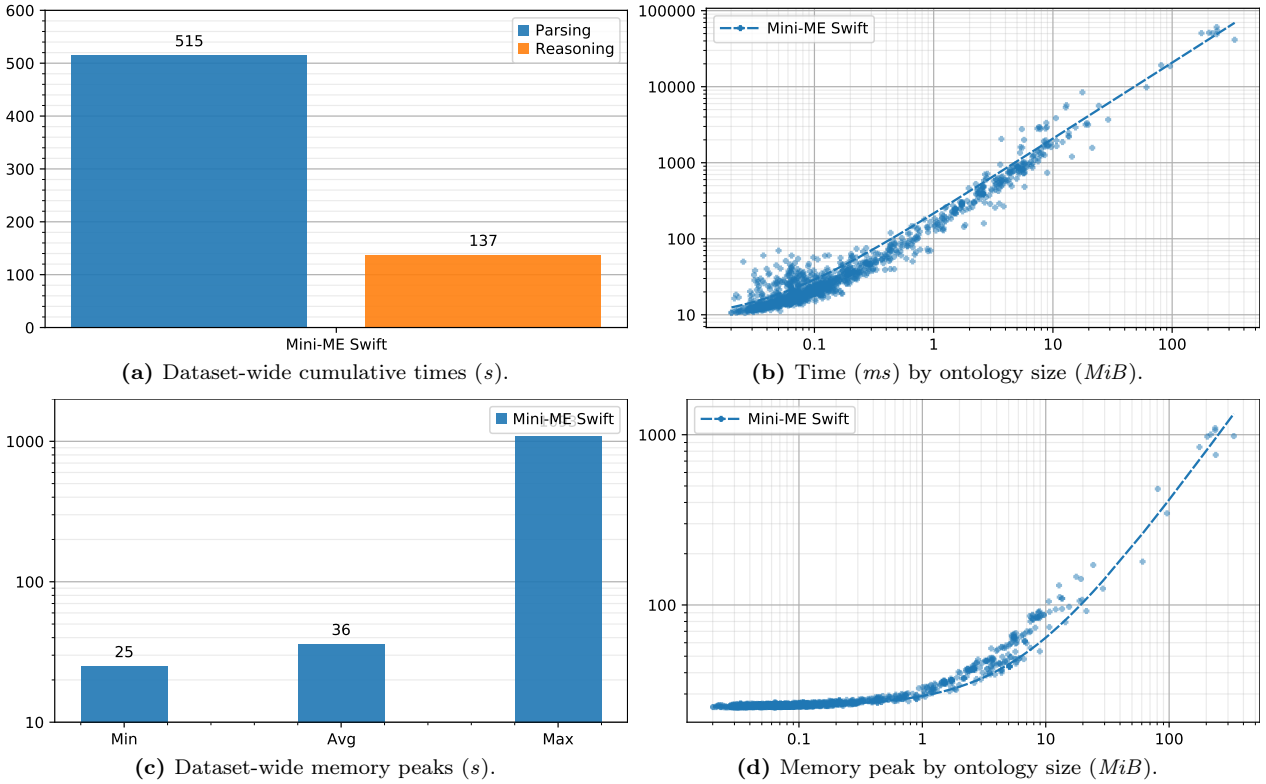
(c) Dataset-wide memory peaks (s).

(d) Memory peak by ontology size (MiB).

Figure 4.11: Classification performance tests on iOS.

- Tables 4.10 and 4.11 reports macOS tests results in a tabular form;

Fact++ and HermiT exhibit a significantly smaller energy footprint for consistency than classification, while each of the other reasoners behave more similarly across the two inferences, consistently with previous findings. Tables 4.12 and 4.13 are generated analogously to Table 4.7 from the previous test case, reporting the time-energy, memory-energy and time-memory Pearson correlation coefficients for each reasoner. They are quite consistent, even though referring to different operating systems and energy profilers. In fact, the correlation between *powertop* scores on Linux and *powermetrics* measurements on macOS, reported in Table 4.14, is quite high. Comparing Table 4.7 with Tables 4.12 and 4.13, it appears that for larger ontologies HermiT has correlation values more similar to those of Fact++. This may mean lower energy-time and energy-memory correlations are a byproduct of lower



(a) Dataset-wide cumulative times (s).

(b) Time (ms) by ontology size (MiB).

(c) Dataset-wide memory peaks (s).

(d) Memory peak by ontology size (MiB).

Figure 4.12: Consistency performance tests on iOS.

time-memory correlation, which depends on the interplay between inference algorithms and the ontology size and constructs. Further investigations are left to more extensive experimental campaigns.

Based on this case study as well as on early experiences by thesis students and interns of the research group laboratory, using EVOWLUATOR to perform tests on real-world reasoners appears to be generally straightforward, while providing a good degree of flexibility. The integration of six desktop reasoners and five additional configurations for mobile tests has required only about 250 lines of Python code. The visualization functionality has come in particularly handy, as the summaries and plots it provides allow for a quick at-a-glance performance comparison.

Table 4.5: ORE 2014 - Summary of classification energy footprint tests on macOS.

Reasoner	Min energy	Avg energy	Max energy
Fact++	475.44	1406.19	7533.51
HermiT	274.50	1273.30	3588.50
Konclude	47.48	149.18	408.60
Mini-ME	141.93	9547.86	55859.39
Mini-ME Swift	55.74	169.42	445.29
TrOWL	208.21	1174.35	3582.45

Table 4.6: ORE 2014 - Summary of consistency energy footprint tests on macOS.

Reasoner	Min energy	Avg energy	Max energy
Fact++	332.82	624.91	1241.54
HermiT	269.42	545.51	1115.08
Konclude	37.33	79.82	184.28
Mini-ME	143.05	7655.57	56533.58
Mini-ME Swift	55.75	137.92	373.81
TrOWL	228.31	987.98	2987.28

Table 4.7: ORE 2014 - Correlation between time, memory peak and energy footprint score on macOS.

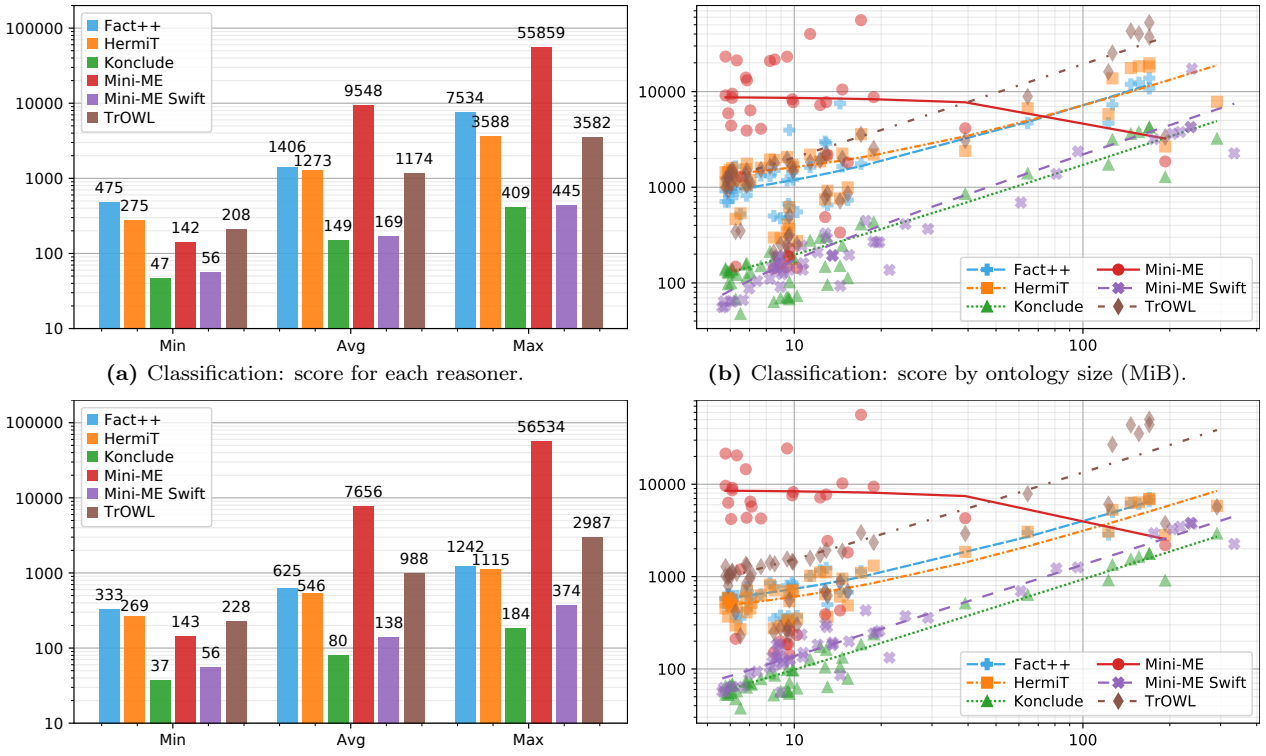
Reasoner	Energy-Time	Energy-Memory	Time-Memory
Fact++	0.98	0.27	0.19
HermiT	0.99	0.96	0.97
Konclude	1.00	0.99	0.99
Mini-ME	1.00	0.28	0.24
Mini-ME Swift	1.00	0.93	0.93
TrOWL	0.99	0.97	0.98

Table 4.8: BioPortal - Summary of classification energy footprint tests on Linux.

Reasoner	Min energy	Avg energy	Max energy
Fact++	23.87	1606.24	6747.2
HermiT	32.65	236.07	1244.58
Konclude	3.61	21.78	56.34
TrOWL	30.7	80.47	171.51

Table 4.9: BioPortal - Summary of consistency energy footprint tests on Linux.

Reasoner	Min energy	Avg energy	Max energy
Fact++	21.77	45.8	94.99
HermiT	20.51	62.66	133.28
Konclude	2.76	16.75	51.4
TrOWL	26.5	79.65	162.73



(a) Classification: score for each reasoner.

(b) Classification: score by ontology size (MiB).

(c) Consistency: score for each reasoner.

(d) Consistency: score by ontology size (MiB).

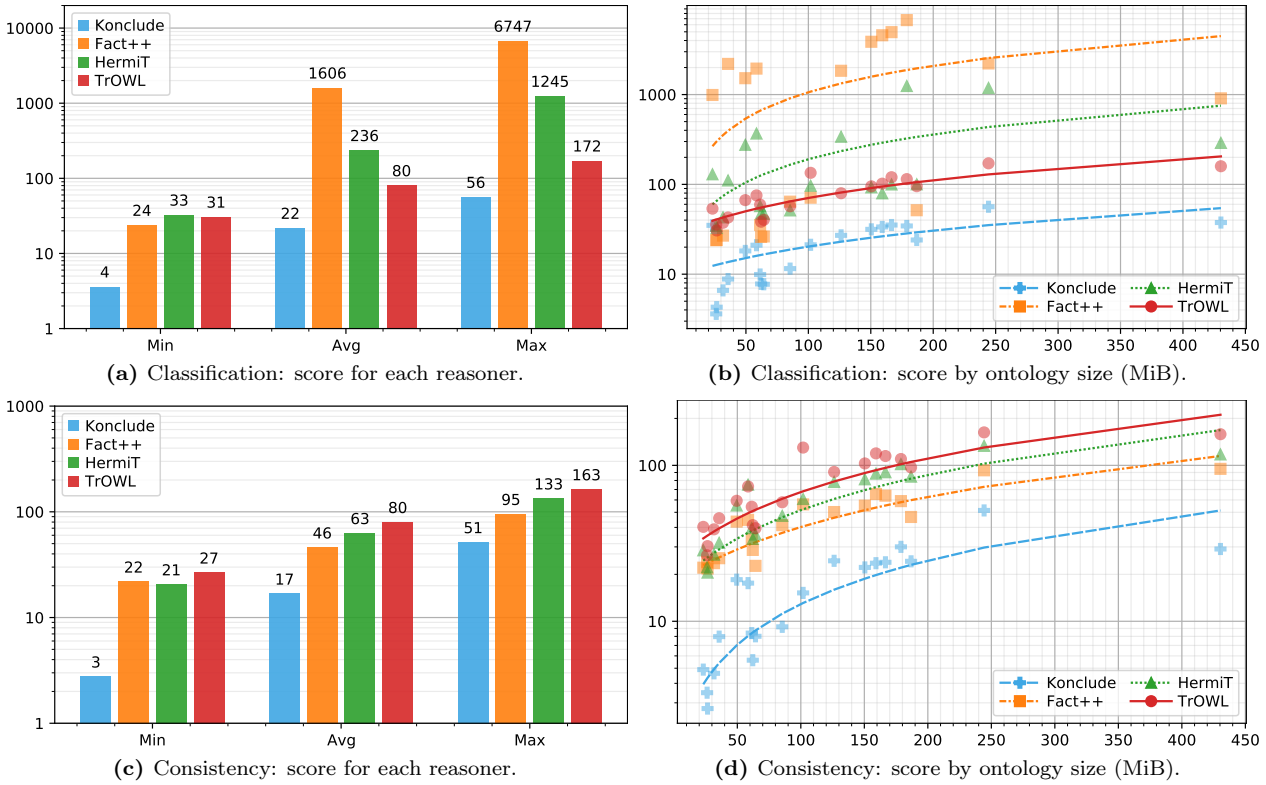
Figure 4.13: ORE 2014 - Energy footprint tests on macOS.

Table 4.10: BioPortal dataset - Summary of classification energy footprint tests on macOS.

Reasoner	Min energy	Avg energy	Max energy
Fact++	1472.49	131134.95	563359.07
HermiT	2524.55	17081.62	95200.06
Konclude	395.70	2031.57	5641.29
TrOWL	2442.32	6483.57	13857.09

Table 4.11: BioPortal dataset - Summary of consistency energy footprint tests on macOS.

Reasoner	Min energy	Avg energy	Max energy
Fact++	1282.61	2949.97	6136.69
HermiT	1436.54	4305.14	9269.16
Konclude	243.23	1703.41	4976.11
TrOWL	1991.29	5736.65	12179.98



(a) Classification: score for each reasoner.

(b) Classification: score by ontology size (MiB).

(c) Consistency: score for each reasoner.

(d) Consistency: score by ontology size (MiB).

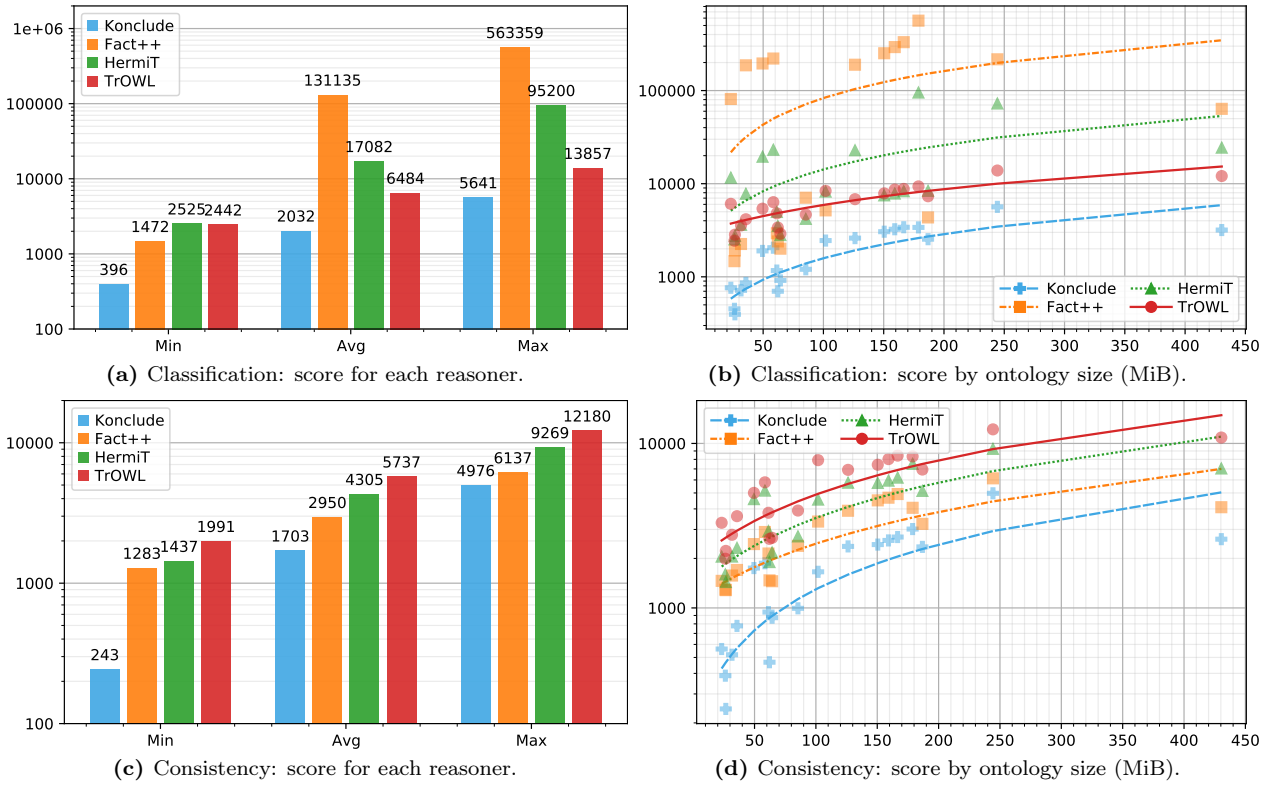
Figure 4.14: BioPortal Energy footprint tests on Linux desktop.

Table 4.12: BioPortal - Correlation between time, memory peak and energy footprint score on Linux.

Reasoner	Energy-Time	Energy-Memory	Time-Memory
Fact++	0.61	0.70	0.42
HermiT	0.69	0.53	0.36
Konclude	0.89	0.91	0.97
TrOWL	0.97	0.97	0.92

Table 4.13: BioPortal - Correlation between time, memory peak and energy footprint score on macOS.

Reasoner	Energy-Time	Energy-Memory	Time-Memory
Fact++	0.61	0.63	0.37
HermiT	0.70	0.63	0.69
Konclude	0.99	0.98	0.96
TrOWL	0.98	0.98	0.98



(a) Classification: score for each reasoner.

(b) Classification: score by ontology size (MiB).

(c) Consistency: score for each reasoner.

(d) Consistency: score by ontology size (MiB).

Figure 4.15: BioPortal Energy footprint tests on Mac Mini.

Table 4.14: BioPortal - Correlation between energy scores on Linux and macOS.

Reasoner	Energy Correlation
Fact++	0.97
HermiT	0.99
Konclude	0.90
TrOWL	0.97

Chapter 5

Application case studies

In order to validate the feasibility, versatility and robustness of the proposed SWoE infrastructure across a spectrum of scenarios, a set of carefully selected case studies has been developed, ranging from nano-scale devices in smart cities to real-time embedded systems in unmanned aerial vehicles (UAVs) and wearable devices, up to ubiquitous Web services. Each case study presented in this chapter highlights different aspects of the overall proposal, demonstrating its adaptability, scalability, and usefulness in diverse contexts.

The first case study focuses on a smart city scenario, where the SWoE, through Cowl's minimal memory footprint, enables nano-scale sensing devices disseminated in urban environments to directly take part in a knowledge-based infrastructure. This application enhances urban mobility and territory management by facilitating the collection, processing, and sharing of semantically annotated data, thereby transforming city spaces into more efficient and responsive environments. The second scenario shifts to the realm of unmanned aerial vehicles, where Tiny-ME is used for on-board, real-time inferences. This case highlights the role of the SWoE in enabling autonomous and explainable decision-making in UAVs, crucial for time-sensitive and trustworthy mission supervision. In the third case study, Tiny-ME is integrated into a smartwatch to evaluate asthma symptom severity, leveraging on-board sensors to gather health data. This application highlights how the SWoE can create, expand and reason on personal health knowledge graphs [45], transforming wearables into advanced, explainable health monitoring agents. The final study presents

a Web-based privacy-oriented local event finder, illustrating the suitability of the stack for client-side retrieval and personalization functionalities in Rich Internet Applications (RIA). This scenario demonstrates how the SWoE enhances user experience by providing semantic capabilities and intelligent information processing while preserving user privacy w.r.t. conventional Web and social networking applications. Together, these case studies aim to underscore the versatility of the SWoE, showing its effective application across various domains, from nano-devices to the World Wide Web, highlighting its broad potential and implications in the overall information technology landscape.

5.1 Extending the Web of Things to embedded sensor networks

Smart road management systems are revolutionizing the way traffic is monitored and controlled on highways and urban streets [89, 88]. The proposed case study highlights how Cowl can be used to manage and share data collected from various smart devices in order to enhance road management, improve traffic flow, and increase public safety. In the reference smart city scenario, sketched in Figure 5.1, urban areas are equipped with smart devices embedded in the road surface, consisting of one or more sensors (*e.g.*, accelerometer, temperature, pressure, and vibration sensors), in order to detect traffic intensity [137] and environmental conditions. Road sensors are also able to interact with further sensorized IoT devices, *e.g.*, vehicles' smart tires, pedestrians' smart shoes, smartphones and wearables, and Unmanned Aerial Vehicles (UAVs, a.k.a. drones), in order to gather and share contextual data in real-time through a common communication protocol. For the case study the *Constrained Application Protocol* (CoAP) [21] has been selected, as it represents an application-layer protocol expressly defined for networks of objects with limited computational, memory and bandwidth resources. CoAP is particularly useful in SWoT/SWoE scenarios, such as collaborative sensor networks, where heterogeneous data must be gathered from scattered nodes

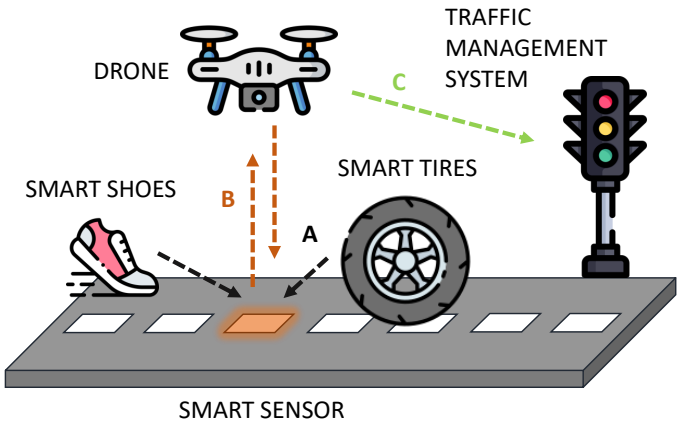


Figure 5.1: Reference scenario for smart road management.

to infer events [102, 99]. The three-step example in Figure 5.1 is described in detail hereafter.

Step A: road surface interactions. As pedestrians and vehicles travel on the equipped road, smart sensors continuously collect data from the road surface. In particular, self-powered triboelectric pressure-velocity nanosensors [133] as well as vibration sensors triggered by the moving vehicles can be used to identify traffic type and intensity by means of Machine Learning algorithms [82]. Detected events and contextual data are periodically updated and annotated within each smart sensor through Cowl. In this way, a multitude of dynamic ubiquitous, geo-referenced knowledge base fragments are created, each containing a set of OWL axioms characterizing the local state of individual road sections in a specific time window. An example of data annotation is shown in Figure 5.2, describing a smart sensing node equipped with a temperature sensor and an accelerometer. *M3-lite* [1] is used as the upper ontology to model device categories, measurement parameters and properties. The *W3C Geospatial* vocabulary [67] is used to specify the geographic location of each node, whereas the *Sensor, Observation, Sample, and Actuator (SOSA)* ontology [53] provides the reference specification for modeling observations and detected events.

Event annotations can be further enriched with data collected by external smart devices passing by a road node. In particular, several manufacturers are

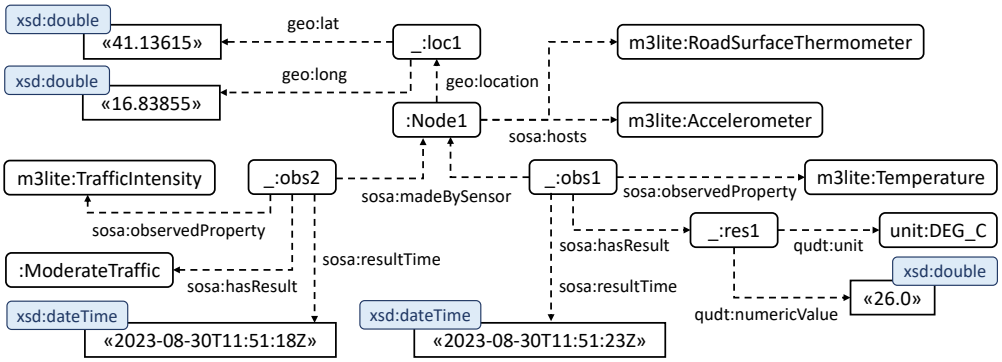


Figure 5.2: Example of road context annotation.

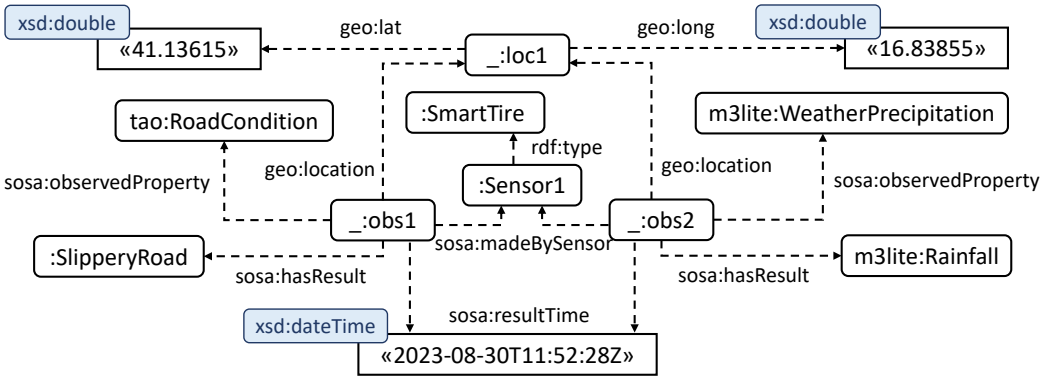


Figure 5.3: Example of annotation received from a smart tire.

currently developing intelligent tires integrating communication functionality for sharing sensor data with both roadside infrastructures and the Electronic Control Unit (ECU) of the car. Smart tires are typically able to identify useful road information (*e.g.*, road surface status, weather conditions, risk of aquaplaning) aiming to prevent critical driving situations. When a vehicle equipped with smart tires approaches a road section with an embedded smart node, the pressure sensor detects the presence of the vehicle and the road node initiates a CoAP-based data exchange to retrieve information from the tire. An example of OWL-based annotation received from a sensorized tire is shown in Figure 5.3, also including concepts related to the transportation domain modeled by the *Transport Administration Ontology (TAO)* [79]. This annotation is sent by the tire upon receiving a CoAP GET request for a dedicated sensor data resource, as showcased in Listing 2. The smart road

```
// 1. Request from the road node to the tire
CON [id=<0xbc90>, token=<0x71>]
GET /sensor_data
```

```
// 2. Response from the tire
ACK [id=<0xbc90>, token=<0x71>]
CODE 2.05 Content
PAYLOAD
...
ObjectPropertyAssertion(sosa:observedProperty _:obs2 m3lite:Rainfall)
ObjectPropertyAssertion(sosa:hasResult _:obs2 _:res1)
ObjectPropertyAssertion(qudt:unit _:res1 unit:MilliM-PER-DAY)
DataPropertyAssertion(sosa:resultTime _:obs2
    "2023-08-30T11:51:23Z"^^xsd:dateTime)
DataPropertyAssertion(qudt:numericValue _:res2 "5.0"^^xsd:double)
...
```

Listing 2: CoAP-based interaction between a road node and a sensorized tire.

node collects data from multiple vehicles and other aforementioned IoT devices as they pass by, then it parses and aggregates information using Cowl in order to store a progressively richer characterization of surrounding traffic flow and road conditions. This information is also fed back to pedestrians and vehicles via CoAP, in order to enrich their own knowledge graphs and enable more accurate real-time inferences for automatic decision.

Step B: road-drone interaction. As shown in Figure 5.1, smart nodes embedded in roads can also interact with urban monitoring drones equipped with high-resolution cameras, environmental sensors, and communication systems. Each drone is dispatched from a designated launchpad and follows a predefined flight path, guided by Global Navigation Satellite Systems (GNSS) and remote control. As the drone hovers above individual road sensors, it receives collected knowledge via CoAP, which includes real-time information on road conditions. At the same time, UAV onboard sensors capture additional contextual information –obtained by processing on-board camera images and from equipped environmental sensors– which can be annotated and shared with road sensors through CoAP messages. An example of this interaction is displayed in Listing 3, where one of the smart road nodes has detected a rain intensity of 5 mm per day. During its flyby, the UAV detects rain intensity of 10 mm per day, and sends a POST request to the road sensor, which updates

```
// 1. Original knowledge graph of the road sensor
...
ObjectPropertyAssertion(sosa:observedProperty _:obs2 m3lite:Rainfall)
ObjectPropertyAssertion(sosa:hasResult _:obs2 _:res1)
ObjectPropertyAssertion(qudt:unit _:res1 unit:MilliM-PER-DAY)
DataPropertyAssertion(sosa:resultTime _:obs2
    "2023-08-30T11:51:23Z"^^xsd:dateTime)
DataPropertyAssertion(qudt:numericValue _:res2 "5.0"^^xsd:double)
...
```

```
// 2. Request from the UAV to the road sensor
CON [id=<0xbc91>, token=<0x72>]
POST /sensor_data
PAYLOAD
...
ObjectPropertyAssertion(sosa:observedProperty _:obs2 m3lite:Rainfall)
ObjectPropertyAssertion(sosa:hasResult _:obs2 _:res1)
ObjectPropertyAssertion(qudt:unit _:res1 unit:MilliM-PER-DAY)
DataPropertyAssertion(sosa:resultTime _:obs2
    "2023-08-30T11:53:15Z"^^xsd:dateTime)
DataPropertyAssertion(qudt:numericValue _:res2 "10.0"^^xsd:double)
...
```

```
// 3. Response from the road sensor
ACK [id=<0xbc91>, token=<0x72>]
2.04 Changed
```

```
// 4. Updated knowledge graph of the road sensor
...
ObjectPropertyAssertion(sosa:observedProperty _:obs2 m3lite:Rainfall)
ObjectPropertyAssertion(sosa:hasResult _:obs2 _:res1)
ObjectPropertyAssertion(qudt:unit _:res1 unit:MilliM-PER-DAY)
DataPropertyAssertion(sosa:resultTime _:obs2
    "2023-08-30T11:53:15Z"^^xsd:dateTime)
DataPropertyAssertion(qudt:numericValue _:res2 "7.5"^^xsd:double)
...
```

Listing 3: CoAP-based interaction between the UAV and a road node.

its own reading by aggregating the two values (in this example, by averaging them).

Step C: drone-traffic management system interaction. Finally, when the UAV enters the communication range of a road-side unit belonging to the distributed urban traffic management system (TMS), it transmits the gathered knowledge for real-time inference about urban conditions. An example of this interaction is reported in Listing 4, where the drone issues a CoAP PUT request to one of the TMS units, resulting in the creation of new records in the system's geographic database (*e.g.*, a PostGIS[1] instance), exposed as a CoAP endpoint. The drone is able to retrieve the geographic coordinates of the collected readings by using Cowl to query its internal knowledge graph: all collected information –concerning *e.g.*, traffic level and type, road surface integrity and wetness, weather conditions– are extracted and sent. Capable TMS devices can then execute more sophisticated (and computationally expensive) inferences and analytics on the overall data, generating reports and alerts which are then shared with relevant city department policy-makers, emergency responders, and connected vehicles. In this way, the traffic flow can be optimized by acting on connected traffic lights and signals, leading to reduced congestion and improved road safety, and public authorities can dispatch emergency services immediately if needed.

```
// 1. Request from the UAV to the TMS
CON [id=<0xbc92>, token=<0x73>]
PUT /geo_data/41.13615/16.83855
PAYLOAD
...
ObjectPropertyAssertion(sosa:observedProperty _:obs1 m3lite:TrafficIntensity)
ObjectPropertyAssertion(sosa:hasResult _:obs1 :ModerateTraffic)
DataPropertyAssertion(sosa:resultTime _:obs1
    "2023-08-30T11:51:18Z"^^xsd:dateTime)
...
```

```
// 2. Response from the TMS
ACK [id=<0xbc92>, token=<0x73>]
2.01 Created
```

Listing 4: CoAP-based interaction between the UAV and a TMS unit.

[1]PostGIS home: `https://postgis.net`

Information collected by road sensors and drones can be also exploited in an Urban Digital Twin [134] control center for long-term analysis and city planning, in order to extract useful insights for optimizing traffic flow distribution and prioritizing preventive maintenance interventions on the road network. These interactions between smart road infrastructures and drones powered by Cowl show the usefulness of combining pervasive data collection and annotation with distributed aerial surveillance to create a comprehensive and flexible urban territorial monitoring system. Ultimately, this can reduce road congestion as well as traffic-related time waste, atmospheric and acoustic pollution, thus improving the overall safety and quality of life for residents.

5.2 Drone autopilot on-board reasoning

Unmanned Aerial Vehicles (UAVs), a.k.a. drones, are increasingly used in diverse fields like search and rescue, precision farming, and logistics, driven by advances in sensor miniaturization and computing technology [109]. Current UAV-related AI applications rely either on ground stations for data analysis, causing latency issues for real-time tasks, or on on-board ML models, adding cost, weight, and energy demand. These methods also lack transparency, affecting trust and accountability in critical scenarios [48]. KRR techniques can offer a solution with their inherent interpretability and logic-based explanations, enhancing UAV situational awareness, autonomous decision-making, and adaptability [101]. The proposed approach involves monitoring UAV internal states and environmental factors, allowing UAVs to adjust operations by comparing current situations with reference states and adapting behavior accordingly.

To validate the feasibility of this approach, sketched in Figure 5.4, a prototypical implementation of devised SWoE tools has been attempted on a popular embedded flight controller platform. Specifically, the Tiny-ME and Cowl libraries have been ported to the *3DR IRIS+*[2] UAV, equipped with a

[2]IRIS+: `https://3dr.com/support/articles/iris/`

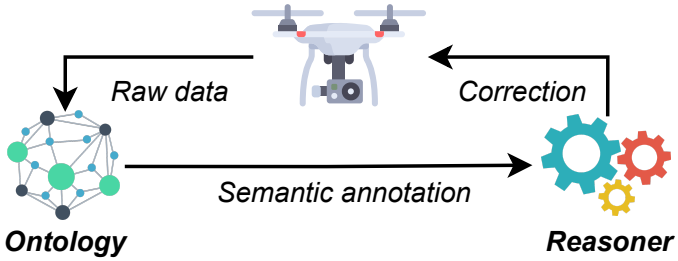


Figure 5.4: Embedded reasoning for autonomous UAV operations.

Pixhawk 1[3] autopilot (*i.e.*, flight controller). This unit has a *STM32F427*[4] system-on-chip, including a 180 MHz ARM Cortex M4 microcontroller and 256 KiB of SRAM. The autopilot runs the *PX4*[5] FMUv2 firmware, based on the *Apache NuttX*[6] real-time operating system (RTOS), supporting the development of user-defined applications and modules.[7] The NuttX/Pixhawk platform has been chosen because it exhibits the typical development challenges of embedded RTOSes, while also introducing very strict constraints on computational resources.

In order to assess on-board reasoning feasibility and usefulness of available inferences and language expressiveness in representative scenarios, two case studies have been envisioned: the first one concerns UAV-based detection of fire and explosion risk from gas or vapor [94], and the second one focuses on on-board context management in a crowd detection and avoidance setting [105]. For both case studies, the reasoner is invoked as part of a periodic task that continuously monitors the environment, collecting internal and external parameters via on-board sensors and OS primitives. The task then constructs a concept expression R representing the current context, and performs semantic matchmaking to compare it with critical scenarios stored in the KB as individuals. For each individual, the UAV checks if it is consistent

[3]Pixhawk 1: `https://docs.px4.io/v1.9.0/en/flight_controller/pixhawk.html`
[4]STM32F427: `https://www.st.com/en/microcontrollers-microprocessors/stm32f427-437.html`
[5]PX4: `https://px4.io`
[6]Apache NuttX: `https://nuttx.apache.org/`
[7]PX4 developer documentation: `https://dev.px4.io`

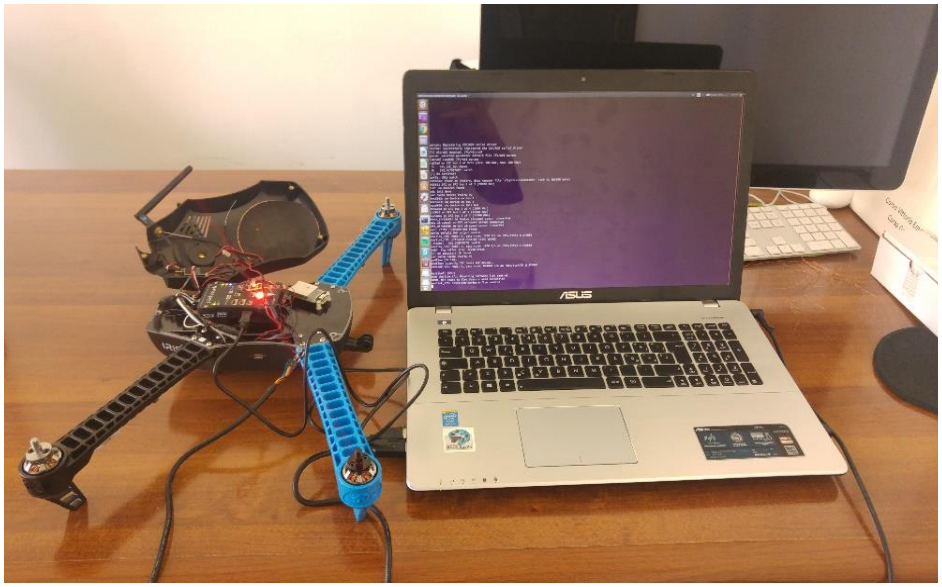


Figure 5.5: UAV testbed setup.

with R and, if the check succeeds, it invokes Concept Abduction to compute the semantic distance d between R and the current individual. In what follows, each individual is considered as a request in the matchmaking framework described in Section 1.2.2, while R is treated as a resource. This entails that the semantic distance d represents "how much" is missing from R to fully align with the reference scenario. If d is below a certain threshold, the individual is a match for the current context R and the behavior of the drone is adapted accordingly.

Environmental hazard detection

According to the European Union (EU) Directive 2014/34/UE,[8] fire and explosion risk from gas or vapor 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

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

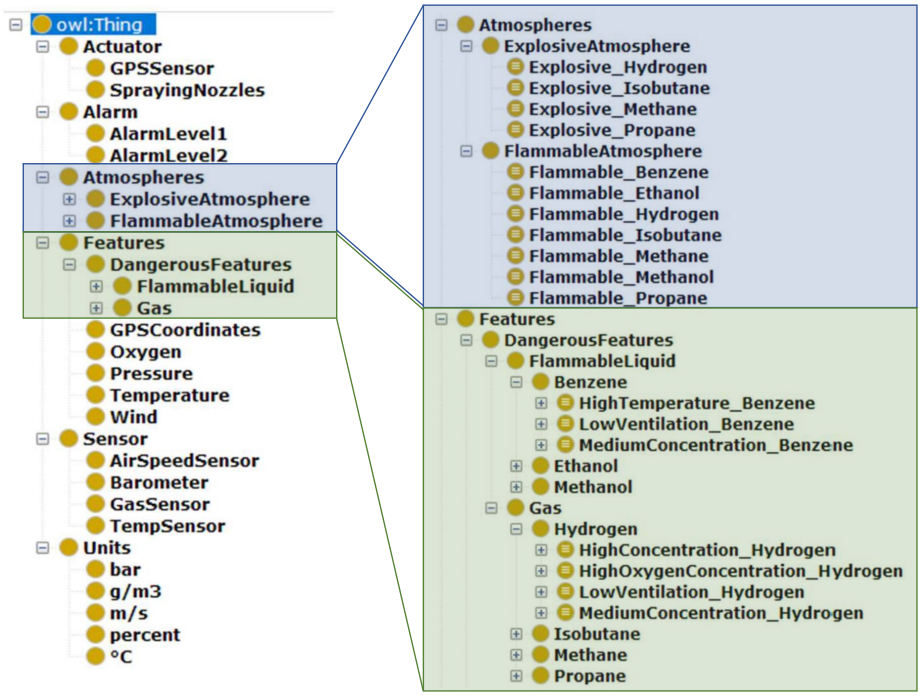


Figure 5.6: Excerpt of the hazard detection knowledge base.

combustion cannot occur; (iii) for a vapor, air temperature is higher than the substance-specific *flashpoint* threshold.

Figure 5.6 shows the upper-level classes of the OWL $\mathcal{ALN}(\mathcal{D})$ KB (292 axioms, 73 classes, 8 object properties, 5 data properties, and 27 individuals) modeled for the case study. 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.

The UAV is endowed with a GNSS antenna as well as sensors for temperature, atmospheric pressure, wind speed, oxygen concentration, and the concentration of each substance to be monitored. As an example, let us consider that on-board sensors detect a 6 g/m^3 methane concentration, 1 m/s wind speed, and 15% oxygen concentration, resulting in the following annotation (reported in Manchester syntax [50]):

```
R: Methane and (hasConcentration only float[>=6.0,<=6.0]) and (withOxy-
genConcentration only (hasConcentration only float[>=15.0,<=15.0])) and
(withWindSpeed only (hasSpeed only float[>=1.0,<=1.0]))
```

The drone performs matchmaking of R with all KB individuals that are instances of `Explosive_Atmosphere` and `Flammable_Atmosphere`, 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. Explosion risk is 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:

```
Explosive_methane: HighConcentration_Methane and
HighOxygenConcentration_Methane and LowVentilation_Methane

Flammable_methane: LowVentilation_Methane and
MediumConcentration_Methane and HighOxygenConcentration_Methane
```

where classes reported in their expressions are defined as:

```
MediumConcentration_Methane ≡ Methane and
(hasConcentration only float[>=6.0,<12.0])

HighConcentration_Methane ≡ Methane and
(hasConcentration only float[>=12.0])

HighOxigenConcentration_Methane ≡ Methane and
(withOxygenConcentration min 1) and (withOxygenConcentration only
(hasConcentration only float[>=14.0]))

LowVentilation_Methane ≡ Methane and (withWindSpeed min 1) and
(withWindSpeed only (hasSpeed only float[<1.0]))
```

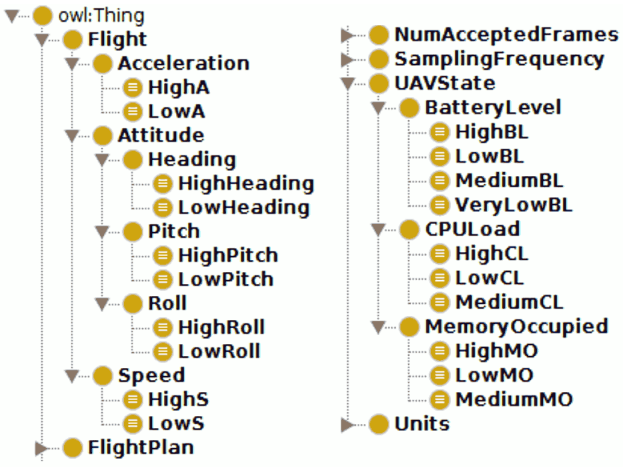


Figure 5.7: Excerpt of the context awareness knowledge base.

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 R has a value of 6 g/m^3. This implies that explosion risk is absent. Conversely, `Flammable_methane` $\sqsubseteq R$ and concept abduction detects a full match ($d = 0$), therefore the semantic distance is below threshold. As a consequence, the UAV raises a fire alert related to the methane substance.

Context awareness

In this second scenario, the UAV is used for crowd detection and avoidance via frame-based image analysis using a nadiral camera payload. A prototypical ontology (296 axioms, 54 classes, 11 data properties and 9 individuals) has been modeled, comprising classes that describe UAV parameters, and individuals characterizing critical scenarios that prevent getting reliable image frame acquisition or timely processing. An excerpt of the ontology is shown in Figure 5.7. To determine if the current conditions align with any of the critical scenarios, the current state of the UAV, e.g.:

```
R: (hasRoll only float[>=1.5,<=1.5]) and (hasPitch only float[>=0.2,<=0.2])
and (hasHeading only float[>=0.1,<=0.1]) and (hasAcceleration only
float[>=0.5,<=0.5]) and (hasSpeed only float[>=5.0,<=5.0]) and
(hasBatteryLevel only float[>=70.0,<=70.0]) and (hasCPULoad only
float[>=98.0,<=98.0]) and (hasMemoryOccupied only float[>=80.0,<=80.0])
```

is compared with each critical scenario available in the KB via semantic matchmaking, obtaining a ranking of scenario profiles based on semantic similarity. Let us consider the following critical scenarios:

```
Critical_1: HighCL and HighMO
Critical_2: MediumCL and MediumMO and HighSF
Critical_3: VeryLowBL
```

with classes defined as follows:

```
HighCL ≡ CPULoad and hasCPULoad only float[>=80.0]
MediumCL ≡ CPULoad and hasCPULoad only float[>=30.0,<80.0]
HighMO ≡ MemoryOccupied and hasMemoryOccupied only float[>=80.0]
MediumMO ≡ MemoryOccupied and hasMemoryOccupied only float[>=30.0,<80.0]
HighSF ≡ SamplingFrequency and hasSamplingFrequency only float[>=10.0]
VeryLowBL ≡ BatteryLevel and hasBatteryLevel only float[<5.0]
```

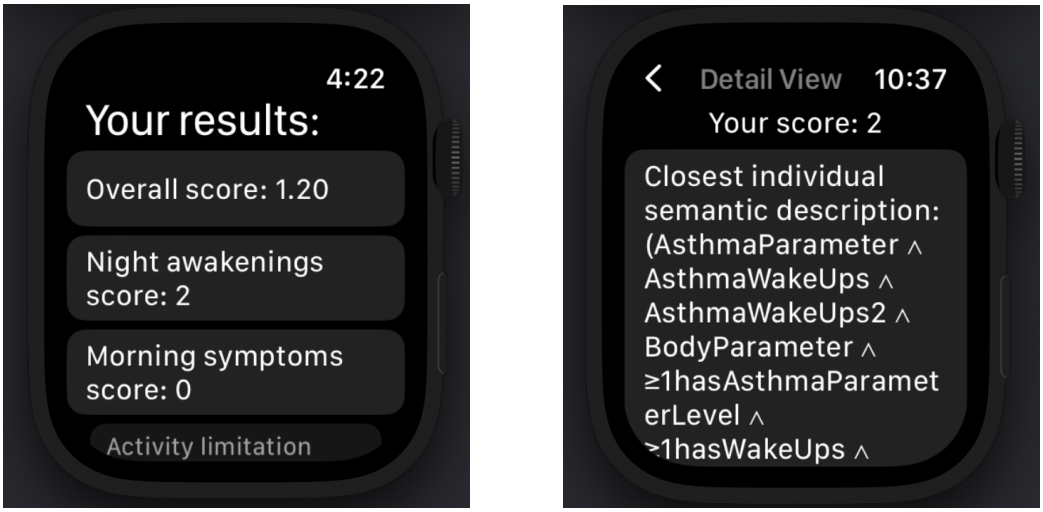
In this particular case, the `Critical_1` individual passes the semantic similarity threshold, due to the currently high CPU load and memory occupancy. To mitigate the impact of the identified critical scenario, a strategic decision is made to skip the processing of the next frame, alleviating the UAV's processing load and enhancing its overall operational efficiency.

5.3 Explainable reasoning on wearables for personal healthcare

In this case study, an Apple Watch[9] application has been developed to assist patients suffering from asthma by estimating the severity of symptoms through on-board automated reasoning. Leveraging capabilities introduced in watchOS[10] version 6, the app has been designed to be independent from external devices, only relying on builtin sensors and processing resources, and to require minimal user intervention. Moreover, providing easy access to logical explanations in such a dependable setting is deemed crucial to increase confidence in inference outcomes. The prototype exploits an experimental port of the Tiny-ME reasoner to the watchOS platform, obtained by cross-compiling its existing Objective-C API (cfr. Section 3.4.1) for Watch devices. The app was tested on an Apple Watch Series 6, the first model to support blood oxygen saturation levels.

The World Health Organization defines[11] chronic asthma as a lung disease affecting people of all ages, caused by inflammation of the airways and contraction of surrounding muscles. Symptoms can vary in intensity and frequency, including cough, wheezing, shortness of breath, and chest tightness. Identifying a precise cause for the onset of the disease is often challenging, though risk factors are well known, and include family history, exposure to pollutants and passive tobacco smoke, prolonged contact with dust, chemicals, and conditions of overweight and obesity. As asthma cannot be cured, international guidelines for treatment focus on controlling the disease, minimizing symptoms and bronchoconstriction. To this aim, the *Asthma Control Questionnaire* (ACQ) [54] has been designed to evaluate the degree of control an individual has over their asthma. It encompasses a series of questions that assess various aspects of the disease, such as the frequency and severity of symptoms, limitation of daily activities, and the patient's use of short-acting

[9] `https://www.apple.com/watch`
[10] `https://www.apple.com/watchos`
[11] `https://www.who.int/news-room/fact-sheets/detail/asthma`



(a) Home screen of the app. **(b)** Semantic-based score explanation.

Figure 5.8: WatchOS application prototype.

bronchodilators. Additionally, it incorporates clinical data through spirometry test results. Each item in the questionnaire is scored on a scale from 0 to 6, where higher scores indicate more severe symptoms or a greater need for medication.

An Apple Watch application prototype, depicted in Figure 5.8, has been developed to estimate responses to the first five ACQ questions without any user input, according to recognized health research correlations [37, 108, 111, 117] with data that can be autonomously collected by the watch through on-board sensors. Health data and sensor readings accessible through the *HealthKit*[12] framework are used to construct an ontology-based annotation of the patient's health status. Subsequently, semantic matchmaking is employed to estimate answers to the following ACQ questions:

1. "*On average, during the past week, how often were you woken by your asthma during the night?*" **(nighttime awakenings)**: the score is computed by counting the number of HealthKit `sleepAnalysis` samples marked as `awake` that overlap with those marked as `inBed`, focusing only on short awakenings followed by sleep.

[12] `https://developer.apple.com/documentation/healthkit`

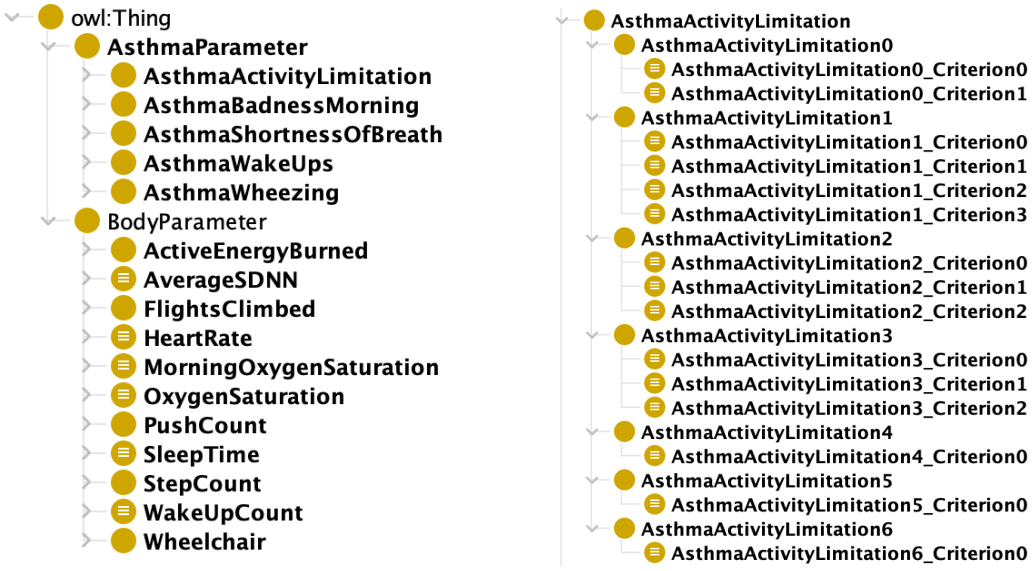
2. "*On average, during the past week, how bad were your asthma symptoms when you woke up in the morning?*" **(morning symptoms severity)**: estimated through the average heart rate and oxygen saturation recorded by Apple Watch from one hour before to two hours after waking up. These averages are categorized into severity ranges from normal to very severe, and are known [111] to be positively correlated to the aggravation of asthma symptoms.

3. "*In general, during the past week, how limited were you in your activities because of your asthma?*" **(activity limitation)**: estimated by querying the averages of `activeEnergyBurned`, `stepCount`, `flightsClimbed` (or `pushCount` for wheelchair users), `heartRateVariabilitySDNN`, and total sleep hours. SDNN stands for *Standard Deviation Normal-to-Normal* and is a well-known method [108] to measure heart rate variability in the time domain as the standard deviation of beat-to-beat intervals. Gathered parameters are compared against a two-week historical average stored by the app. A current week's average that is at least 10% lower than the historical average indicates activity limitation.

4. "*In general, during the past week, how much shortness of breath did you experience because of your asthma?*" **(dyspnea severity)**: linked to the average `heartRateVariabilitySDNN` over the last week. This parameter is known [37] to be linked to dyspnea. A scale based on health research [108] categorizes the average into healthy, compromised health, and unhealthy ranges.

5. "*In general, during the past week, how much of the time did you wheeze?*" **(wheezing)**: associated with the average weekly blood oxygen level. The average of the past week's `oxygenSaturation` samples is compared against a scale to estimate the severity of wheezing, as low oxygen saturation levels are known to be correlated to more severe asthma symptoms [117], including wheezing.

The app displays an overall health score for the patient, computed as the average of the scores of each ACQ question (Figure 5.8a). For each score, a



(a) Top-level taxonomy **(b)** Asthma parameter levels and criteria

Figure 5.9: Excerpt of the asthma knowledge base.

detailed semantic-based explanation can be displayed, showing the criteria that led to its attribution. Figure 5.8b showcases a prototypical version of the detail view, where explanations are provided in textual form. A more user-friendly solution would summarize outcomes through pictorial elements, such as icons and other graphical controls.

The user profile annotation is built by querying HealthKit for specific data samples, aggregating them over a week's period, and then mapping values to appropriate OWL constructs. A specific $\mathcal{ALN}(\mathcal{D})$ ontology has been modeled, describing the user's body parameters such as heart rate, oxygen saturation, number of steps, etc. as subclasses of the `BodyParameter` class, as shown in Figure 5.9a. Responses to ACQ questions have been modeled as subclasses of `AsthmaParameter`, with leaf concepts representing criteria for the attribution of a specific score to each question (Figure 5.9b). Each leaf class is then associated to a KB individual, used as a resource for matchmaking purposes.

As an example, consider the following user profile annotation:

User_Profile:
AverageActiveEnergyBurned and AverageFlightsClimbed and AverageStepCount and (not Wheelchair) and (hasHeartRate only int[>=80,<=80]) and (hasMorningOxygenSaturation only float[>=95.0,<=95.0]) and (hasOxygenSaturation only float[>=90.0,<=90.0]) and (hasSDNN only int[>=105,<=105]) and (hasSleepTime only float[>=42.0,<=42.0]) and (hasWakeUps only int[>=8,<=8])

A semantic matchmaking process is initiated for each ACQ question, considering only the relevant subset of individuals representing match criteria for each score level. The following individuals match the above user profile:

AsthmaWakeUps2_Criterion0:
(hasAsthmaParameterLevel only int[>=2,<=2]) and (hasWakeUps min 1) and (hasWakeUps only int[>=6,<=10])

AsthmaBadnessMorning0_Criterion0:
(hasAsthmaParameterLevel only int[>=0,<=0]) and (hasHeartRate min 1) and (hasHeartRate only float[<90.0]) and (hasMorningOxygenSaturation min 1) and (hasMorningOxygenSaturation only float[>=93.0])

AsthmaActivityLimitation2_Criterion2:
(hasAsthmaParameterLevel only int[>=2,<=2]) and BelowAverageStepCount and (not Wheelchair) and (hasSDNN min 1) and (hasSleepTime min 1) and (hasSDNN only int[>=100]) and (hasSleepTime only int[>=42,<=48])

AsthmaShortnessOfBreath0_Criterion0:
(hasAsthmaParameterLevel only int[>=0,<=0]) and (hasSDNN min 1) and (hasSDNN only int[>=100])

AsthmaWheezing2_Criterion0:
(hasAsthmaParameterLevel only int[>=0,<=0]) and (hasOxygenSaturation min 1) and (hasOxygenSaturation only float[>=89.0,<=91.0])

Note how each individual has a `hasAsthmaParameterLevel` datatype restriction, whose value represents the exact score for the corresponding ACQ question. Due to the above matches, the reasoning process results in the following scores:

- Night awakenings: 2;
- Morning symptoms: 0;
- Activity limitation: 2;
- Shortness of breath: 0;
- Wheezing: 2.

and an overall score of 1.2, indicating a low level of incidence of (*i.e.*, a good level of control on) the disease.

Although clinical validation has not been performed yet for the overall proposed ACQ estimation methodology, from a SWoE perspective the case study demonstrates that stand-alone apps on wearable devices are able to collect, annotate and reason upon sensor data in order to provide health recommendations associated with meaningful explanations [130]. Enabling such capabilities, without resorting to more powerful companion devices for semantic-enabled processing, preserves data privacy and discloses further potential solutions based on a *biofeedback* loop for improving user's health and well-being through increased situation awareness in several scenarios.

5.4 Privacy-conscious (mobile) Web

In order to demonstrate the practicality and benefits of the proposed SWoE technological stack in Web contexts, a case study has been developed on the retrieval and preference-based ranking of local events. In modern Web platforms, users often need to disclose their profile information to receive

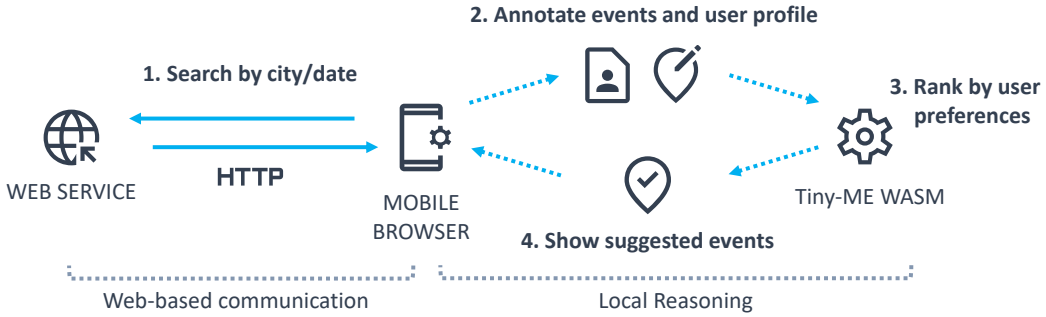


Figure 5.10: Architecture and workflow of the proposed Web application.

personalized recommendations for events, products, and other services. The proposed workflow, illustrated in Figure 5.10, leverages the WebAssembly port of the Tiny-ME reasoner –described in Section 3.4.3– to provide an alternative general-purpose approach, based on four main steps:

1. a mobile Web application gathers non-personalized available information from the Web;
2. this information is then annotated in accordance with an OWL domain ontology;
3. the annotated data is used to pinpoint resources that align closely with the user's profile and preferences;
4. the results are subsequently displayed in the user interface, allowing for user selection and/or query refinement.

A key aspect of the approach is that only the first step involves querying a remote server, while all subsequent processing is performed locally on the user's mobile device. This approach not only ensures privacy-preserving information management, but also eliminates the delays typically associated with interactions with a remote reasoning engine.

Let us consider the following example: *Martina, while on vacation in San Francisco, is interested in purchasing a ticket for a musical event, spending*

no more than $80.00. She is a fan of alternative rock bands and would like to attend a tour date rather than a one-time concert.

The framework, as detailed in Figure 5.10, allows users to access information about local events through a Web application on their mobile browser. The data for this case study is sourced from *Ticketmaster*[13] via their public RESTful API.[14] Initially, as shown in Figure 5.11a, users retrieve a list of nearby events based on basic parameters like city and date range. This data is generic and not tailored to individual preferences. To personalize this information, users can select various features and preferences to create their private personal profile (Figure 5.11b) on-device, which is then translated into an ontology-based semantic annotation. Figure 5.12 illustrates the TBox concepts, which correspond to the categories provided by the service API. The specific details of the selected preferences are displayed below:

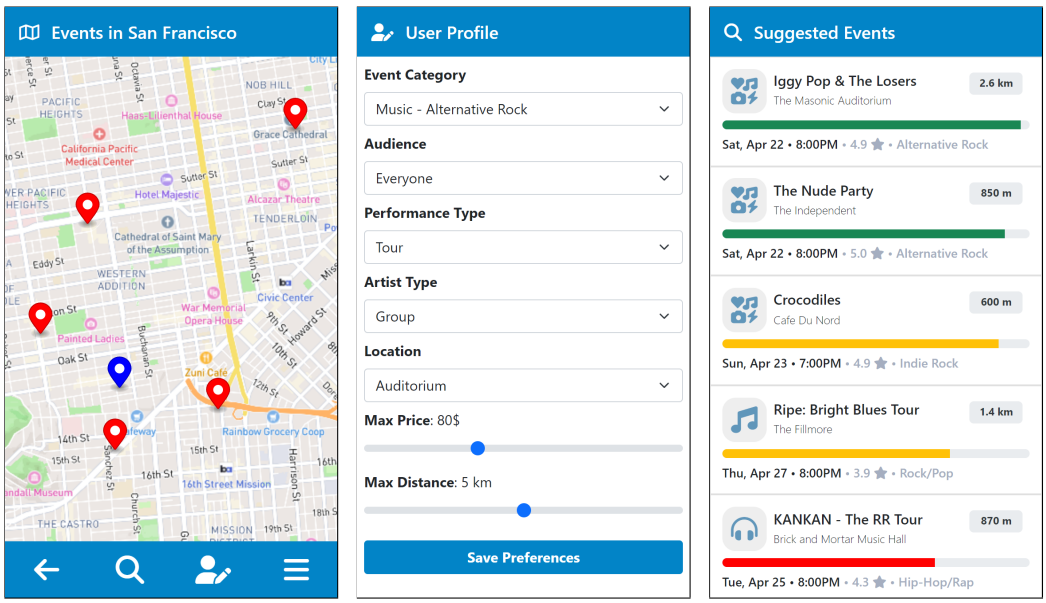
User_Profile: Event and (hasAudience only EveryOne) and (hasPrice only float[<=80.0]) and (hasStyle only Tour) and (hasCategory only Alternative_Rock) and (hasType only Group) and (hasCategory min 1) and (hasStyle min 1) and (hasAudience min 1)

After a pre-filtering step based on a maximum distance of 5 km from the mobile device location and considering only events occurring in the current week, the user profile is compared via semantic matchmaking with the following semantically annotated event descriptions:

Crocodiles: Event and (hasAudience only EveryOne) and (hasPrice only float[>=163.0,<=180.0]) and (hasStyle only Tour) and (hasType only Band) and (hasCategory only Indie_Rock)

Iggy_Pop_and_The_Losers: Event and (hasAudience only EveryOne) and (hasPrice only float[>=50.0,<=100.0]) and (hasStyle only Tour) and (hasType only Band) and (hasCategory only Alternative_Rock)

[13] https://www.ticketmaster.com
[14] https://developer.ticketmaster.com/products-and-docs/apis/getting-started/



(a) Available events. **(b)** User profile. **(c)** Suggested events.

Figure 5.11: Screenshots of the Web application.

KANKAN-The_RR_Tour: Event and (hasAudience only EveryOne) and (hasPrice only float[>=25.0,<=105.0]) and (hasStyle only Concert) and (hasType only Performer) and (hasCategory only Hip_Hop_Rap)

Ripe-Bright_Blues_Tour: Event and (hasAudience only EveryOne) and (hasPrice only float[>=27.0,<=27.0]) and (hasStyle only Tour) and (hasType only Band) and (hasCategory only Pop_Rock)

The_Nude_Party: Event and (hasAudience only EveryOne) and (hasPrice only float[>=20.0,<=20.0]) and (hasStyle only Concert) and (hasType only Band) and (hasCategory only Alternative_Rock)

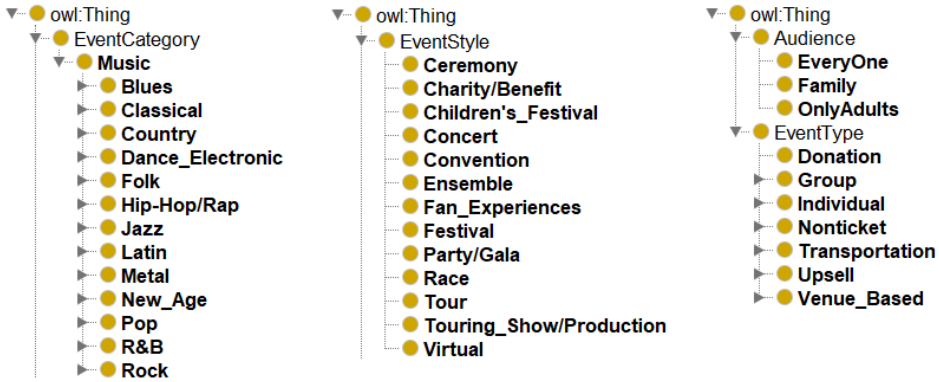


Figure 5.12: Reference ontology for the event finder application.

The returned list of events, shown in Figure 5.11c, is ranked according to the following *utility function*:

$$u(R,C) = 100[1 - \frac{s\_penalty(R,C)}{s\_penalty(R,\top)}(1 + \frac{distance(R,C)}{max\_distance})]$$

where $s\_penalty(R,C)$ is the result of the semantic matchmaking task and it represents the semantic distance between user profile R and event annotation C, normalized by $s\_penalty(R,\top)$; the latter is the semantic distance between R and *top*, and it depends only on the ontology structure. The $distance(R,C)$ context parameter is included to exploit the event geographical distance as weighting factor. The utility function also converts the semantic distance value into a more intuitive percentage score, as shown in Table 5.1. It can be observed that if $s\_penalty(R,C) = 0$ then the final score is 100%, regardless of the geographic distance. Based on user preferences and event descriptions, the most suitable event, as listed in Figure 5.11c, is the individual `Iggy_Pop_and_The_Losers`, even if its physical distance is greater than the other available resources.

Table 5.1: Musical events ranked by the utility function.

Musical events	***s_penalty(R,C)***	***distance(R,C)***	***u(R,C)***
Iggy Pop & The Losers	0.20	2.60 km	97.97 %
The Nude Party	1.00	0.85 km	93.22 %
Crocodiles	1.51	0.60 km	89.81 %
Ripe Bright Blues Tour	4.00	1.40 km	72.59 %
KANKAN - The RR Tour	6.24	0.87 km	57.68 %

Conclusion and perspectives

The Semantic Web of Everything naturally embodies the ongoing progression of the Internet of Things towards the Internet of Everything, and enables meaningful logic-based interactions among its actors. Its materialization requires novel Semantic Web architectures and tools, able to permeate all scales of computing, from capable cloud infrastructures to severely resource-constrained micro- and nano-devices.

This dissertation has addressed the design and implementation of a practical architecture that meets the peculiar requirements of the SWoE vision. The proposed technology stack, comprising Cowl for access and manipulation of knowledge, Tiny-ME for pervasive reasoning, and evOWLuator for systematic assessments, demonstrates a comprehensive approach towards realizing the Semantic Web of Everything, with each component playing a key role in bridging the gap between its theoretical aspects and practical application in diverse computing environments.

Cowl showcases it is possible to leverage semantic technologies in severely restricted settings by means of novel techniques and targeted architectural choices and optimizations, while remaining versatile enough for high performance access to knowledge graphs on more capable platforms. *Tiny-ME* offers a robust reasoning framework on a moderately expressive fragment of OWL 2, operating effectively across a spectrum of platforms, from containerized cloud (micro)services to smaller edge devices and embedded boards. This flexibility ensures that Tiny-ME can provide the necessary reasoning capabilities for the SWoE, at each scale of device computational resources. Finally, *evOWLuator* complements the stack by providing a systematic framework for evaluating the performance and efficiency of Semantic Web technologies. Its focus on aspects like energy consumption and remote inference capability is crucial

for assessing the suitability of these technologies in a SWoE context, where resource efficiency is paramount. This infrastructure lays the groundwork for future advancements in the field and further refinement of the developed technologies, paving the way for more intelligent, interconnected, and autonomous systems within the realm of the Internet of Everything.

For Cowl, future work involves broadening its capabilities to include support for parsing and serializing RDF-based OWL syntaxes, such as RDF/XML and Turtle. This expansion aims to enhance interoperability with a wider range of existing Semantic Web tools and datasets. Additionally, the integration of inference capabilities through interfaces compatible with established OWL reasoners is planned. Finally, Cowl's low memory footprint and its unique ability to process ontologies as streams of axioms enables the development of a large-scale cross-platform distributed knowledge graph framework. Such a framework could enable nodes with minimal processing power to participate in creating and using semantically annotated information, fostering truly collaborative, pervasive, knowledge-aware applications.

In the ongoing development of Tiny-ME, the focus will be on enhancing language expressiveness and extending its reasoning capabilities. The primary goal is to support reasoning in OWL 2 $\mathcal{EL}$, $\mathcal{EL}^+$, and $\mathcal{EL}^{++}$, which necessitates not only the expansion of data structures but also the development of new reasoning algorithms in its C core. Thanks to its novel architectural approach, these advancements in inference capabilities will require a one-time implementation effort, with all associated APIs transparently benefiting from them. Additionally, Tiny-ME's plug-in datatype architecture will be exploited to introduce non-standard datatypes, thereby broadening its applicability to a wider range of practical domains. This addition will enhance the versatility and adaptability of the system, making it able to meet the diverse and evolving needs within the Semantic Web of Everything. A further research direction involves integrating the proposed KRR infrastructure with Large Language Models, in an effort to improve their logic-based inference capabilities.

Lastly, future improvements are planned for evOWLuator to substantially augment its functionality and user experience. First of all, a systematic

survey of actively developed reasoning engines for desktop and mobile systems is planned, focusing on energy footprint evaluation. Furthermore, evOWLuator's visualization capabilities will be extended, introducing a wider array of visualization types and options, significantly increasing its usefulness in academic research and data analytics contexts. By offering more sophisticated and diverse data representation tools, the framework will provide deeper insights and more intuitive analysis of the performance of semantic web tools. Additionally, the development of a Web-based front-end will streamline the process of running evaluations. This user-friendly interface will make evOWLuator more accessible and convenient, facilitating its use across various platforms and significantly enhancing its appeal to a broader user base.

In advancing the SWoE vision, the transition from academic contributions to applied technologies ready for production use is of utmost importance. This evolution requires not only rigorous validation and refinement of the proposed frameworks within real-world scenarios, but also fruitful engagements with both the research and industry communities. The journey toward realizing the SWoE vision involves perfecting the proposed case studies, demonstrating their relevance and applicability in industrially relevant environments, and fostering partnerships that facilitate technology transfer and collaborative innovation.

A critical aspect of this progression is the improvement of the *Technology Readiness Level* (TRL) of the proposed solutions, which is essential for ensuring their practicality and effectiveness in real-world settings. This effort necessitates collaboration initiatives with partners in industry and academia to align the technologies with market and research needs, integrate them into existing ecosystems, and address the challenges faced by different vertical sectors and research fields. Such collaborations are invaluable for accelerating the adoption of the novel technologies and paradigms, and for their integration into the broader IoT/IoE landscape.

For this reason, the success of the SWoE hinges on robust community outreach. The initiatives undertaken thus far, including the public release of tools and their source code under permissive licenses, along with curated

documentation, have initiated promising dialogues with scientists and practitioners from both academic and industrial backgrounds. Moving forward, it is crucial to intensify these efforts, engaging more deeply with the community to solicit feedback, foster collaboration, and facilitate the exchange of ideas. This will not only result in refinements and enhancements to the existing methods and solutions, but may also inspire the development of new tools, frameworks, and applications that build upon the groundwork laid out in this dissertation. Hopefully, this might pave the way for a future where semantic technologies can transform the fabric of the IoT into a more pervasive, intelligent, interconnected, and accessible domain.

Bibliography

[1] Rachit Agarwal, David Gomez Fernandez, Tarek Elsaleh, Amelie Gyrard, Jorge Lanza, Luis Sanchez, Nikolaos Georgantas, and Valerie Issarny. Unified IoT ontology to enable interoperability and federation of testbeds. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pages 70–75, 2016.

[2] Safdar Ali and Stephan Kiefer. μOR–A micro OWL DL reasoner for ambient intelligent devices. In *4th International Conference on Advances in Grid and Pervasive Computing*, pages 305–316. Springer, 2009.

[3] Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the EL envelope. In *International Joint Conference on Artificial Intelligence*, volume 5, pages 364–369, 2005.

[4] Franz Baader, Diego Calvanese, Deborah L McGuinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2002. 2nd Ed.

[5] Franz Baader, Bernhard Hollunder, Bernhard Nebel, Hans-Jürgen Profitlich, and Enrico Franconi. An empirical analysis of optimization techniques for terminological representation systems. *Applied Intelligence*, 4(2):109–132, 1994.

[6] Samantha Bail, Birte Glimm, Ernesto Jimenez-Ruiz, Nicolas Matentzoglu, Bijan Parsia, and Andreas Steigmiller. ORE 2014 OWL Reasoner Evaluation Live Competition. `http://dl.kr.org/ore2014`. Accessed: 2023-03-20.

[7] Samantha Bail, Bijan Parsia, and Ulrike Sattler. JustBench: a framework for OWL benchmarking. In *International Semantic Web Conference*, pages 32–47. Springer, 2010.

[8] Christian Becker and Christian Bizer. Exploring the Geospatial Semantic Web with DBpedia mobile. *Journal of Web Semantics*, 7(4):278–286, 2009.

[9] David Beckett. The design and implementation of the Redland RDF application framework. *Computer Networks*, 39(5):577–588, 2002.

[10] Charles Bell. *MicroPython for the Internet of Things*. Springer, 2017.

[11] Alexandre Bento, Lionel Médini, Kamal Singh, and Frédérique Laforest. Do Arduinos Dream of Efficient Reasoners? In Paul Groth, Maria-Esther Vidal, Fabian Suchanek, Pedro Szekley, Pavan Kapanipathi, Catia Pesquita, Hala Skaf-Molli, and Minna Tamper, editors, *The Semantic Web*, pages 289–304, Cham, 2022. Springer International Publishing.

[12] Tim Berners-Lee, Roy Fielding, and Larry Masinter. Uniform Resource Identifiers. RFC 3986, Internet Engineering Task Force, January 2005. `https://rfc-editor.org/rfc/rfc3986.txt`.

[13] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):28–37, 2001.

[14] Ivano Bilenchi, Floriano Scioscia, and Michele Ruta. Cowl: A Lightweight OWL Library for the Semantic Web of Everything. In Agapito *et al.*, editor, *Current Trends in Web Engineering. ICWE 2022.*, pages 100–112, Cham, 2023. Springer.

[15] Ivano Bilenchi, Arnaldo Tomasino, Filippo Gramegna, Saverio Ieva, Agnese Pinto, Giuseppe Loseto, Floriano Scioscia, and Michele Ruta. Knowledge Representation and Reasoning for Unmanned Aerial Vehicle Intelligence. In *7th Italian Workshop on Embedded Systems (IWES 2022)*, 2022.

[16] Tegawendé F. Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillère. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 303–312, 2013.

[17] Bluetooth Special Interest Group. Bluetooth. Specification, Bluetooth Special Interest Group, 1998. `https://bluetooth.com`.

[18] Carlos Bobed, Roberto Yus, Fernando Bobillo, and Eduardo Mena. Semantic reasoning on mobile devices: Do Androids dream of efficient reasoners? *Journal of Web Semantics*, 35:167–183, 2015.

[19] Jürgen Bock, Peter Haase, Qiu Ji, and Raphael Volz. Benchmarking OWL Reasoners. In *ARea2008 – Workshop on Advancing Reasoning on the Web: Scalability and Commonsense*, 2008.

[20] Alexander Borgida. Description logics in data management. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):671–682, 1995.

[21] Carsten Bormann, Angelo P Castellani, and Zach Shelby. Coap: An application protocol for billions of tiny internet nodes. *Internet Computing, IEEE*, 16(2):62–67, 2012.

[22] Carsten Bormann, Mehmet Ersue, and Ari Keranen. Terminology for Constrained-Node Networks. RFC 7228, Internet Engineering Task Force, May 2014.

[23] Tim Bray, Dave Hollander, Andrew Layman, Richard Tobin, and Henry S. Thompson. Namespaces in XML 1.0. Recommendation, W3C, December 2009. `http://www.w3.org/TR/xml-names/`.

[24] Dan Brickley and Ramanathan V. Guha. RDF Schema 1.1. Recommendation, W3C, February 2014. `https://www.w3.org/TR/rdf-schema/`.

[25] Luca Buoncompagni, Syed Yusha Kareem, and Fulvio Mastrogiovanni. Owloop: A modular api to describe owl axioms in oop objects hierarchies. *SoftwareX*, 17:100952, 2022.

[26] James Chamberlain, Corinne Blanchard, Sam Burlingame, Sarika Chandramohan, Eric Forestier, Gary Griffith, Mary Lou Mazzara, Subu Musti, Sung-Ik Son, Glenn Stump, and Christoph Weiss. *IBM WebSphere RFID Handbook: A Solution Guide*. IBM International Technical Support Organization, May 2006.

[27] Connectivity Standards Alliance. ZigBee. Specification, Connectivity Standards Alliance, 2003. `https://csa-iot.org`.

[28] Daniele Dell'Aglio, Emanuele Della Valle, Frank van Harmelen, and Abraham Bernstein. Stream reasoning: A survey and outlook. *Data Science*, 1(1-2):59–83, 2017.

[29] Kathrin Dentler, Ronald Cornet, Annette Ten Teije, and Nicolette De Keizer. Comparison of reasoners for large ontologies in the OWL 2 EL profile. *Semantic Web*, 2(2):71–87, 2011.

[30] Tommaso Di Noia, Eugenio Di Sciascio, and Francesco M. Donini. Semantic matchmaking as non-monotonic reasoning: A description logic approach. *Journal of Artificial Intelligence Research (JAIR)*, 29:269–307, 2007.

[31] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Software-based energy profiling of android apps: Simple, efficient and reliable? In *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*, pages 103–114. IEEE, 2017.

[32] Ergin Dinc, Murat Kuscu, Bilgesu Arif Bilgin, and Ozgur Baris Akan. Internet of Everything: A Unifying Framework Beyond Internet of Things. In *Harnessing the Internet of Everything (IoE) for Accelerated Innovation Opportunities*, pages 1–30. IGI Global, 2019.

[33] Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. Reasoning in description logics. *Principles of Knowledge representation*, 1:191–236, 1996.

[34] Martin Duerst and Michel Suignard. Internationalized Resource Identifiers. RFC 3987, Internet Engineering Task Force, January 2005. https://rfc-editor.org/rfc/rfc3987.txt.

[35] Timofey Ermilov, Norman Heino, and Sören Auer. Ontowiki mobile: knowledge management in your pocket. In *Proceedings of the 20th International Conference Companion on World Wide Web*, pages 33–34, 2011.

[36] César Estébanez, Yago Saez, Gustavo Recio, and Pedro Isasi. Performance of the most common non-cryptographic hash functions. *Software: Practice and Experience*, 44(6):681–698, 2014.

[37] Nicholas D. Giardino, Leighton Chan, and Soo Borson. Combined heart rate variability and pulse oximetry biofeedback for chronic obstructive pulmonary disease: preliminary findings. *Applied psychophysiology and biofeedback*, 29:121–133, 2004.

[38] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. HermiT: an OWL 2 reasoner. *Journal of Automated Reasoning*, 53(3):245–269, 2014.

[39] Rafael Gonçalves, Samantha Bail, Ernesto Jiménez-Ruiz, Nicolas Matentzoglu, Bijan Parsia, Birte Glimm, and Yevgeny Kazakov. OWL reasoner evaluation (ORE) workshop 2013 results. In *ORE*, pages 1–18, 2013.

[40] Filippo Gramegna, Arnaldo Tomasino, Saverio Ieva, Ivano Bilenchi, Agnese Pinto, Giuseppe Loseto, Floriano Scioscia, and Michele Ruta. RideMATCHain: a Semantic-enhanced Blockchain Marketplace for Ridesharing. In *8th Italian Conference on ICT for Smart Cities And Communities (I-CiTies 2022)*, 2022.

[41] Stephan Grimm, Michael Watzke, Thomas Hubauer, and Falco Cescolini. Embedded $\mathcal{EL}$ + Reasoning on Programmable Logic Controllers. In *International Semantic Web Conference*, pages 66–81. Springer, 2012.

[42] Thomas R Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.

[43] Isa Guclu, Yuan-Fang Li, Jeff Z Pan, and Martin J Kollingbaum. Predicting energy consumption of ontology reasoning over mobile devices. In *International Semantic Web Conference*, pages 289–304. Springer, 2016.

[44] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, 2005.

[45] Amelie Gyrard, Manas Gaur, Saeedeh Shekarpour, Krishnaprasad Thirunarayan, and Amit Sheth. Personalized health knowledge graph. In *CEUR workshop proceedings*, volume 2317. NIH Public Access, 2018.

[46] Oliver Hahm, Emmanuel Baccelli, Hauke Petersen, and Nicolas Tsiftes. Operating systems for low-end devices in the Internet of Things: a survey. *IEEE Internet of Things Journal*, 3(5):720–734, 2015.

[47] Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a global data space*. Springer Nature, 2022.

[48] High-Level Expert Group on AI. Ethics guidelines for trustworthy AI. Technical report, European Commission, Brussels, April 2019.

[49] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for OWL Ontologies. *Semantic Web*, 2(1):11–21, 2011.

[50] Matthew Horridge and Peter Patel-Schneider. OWL 2 Web Ontology Language Manchester Syntax (Second Edition). W3C note, W3C, December 2012. `http://www.w3.org/TR/owl2-manchester-syntax`.

[51] Martha Imprialou, Giorgos Stoilos, and Bernardo Cuenca Grau. Benchmarking ontology-based query rewriting systems. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, pages 779–785, 2012.

[52] International Organization for Standardization. C11 Standard, ISO/IEC 9899:2011. Draft, ISO, 2011.

[53] Krzysztof Janowicz, Armin Haller, Simon JD Cox, Danh Le Phuoc, and Maxime Lefrançois. SOSA: A lightweight ontology for sensors, observations, samples, and actuators. *Journal of Web Semantics*, 56:1–10, 2019.

[54] EF Juniper, PM O'byrne, GH Guyatt, PJ Ferrie, and DR King. Development and validation of a questionnaire to measure asthma control. *European respiratory journal*, 14(4):902–907, 1999.

[55] Yong-Bin Kang, Yuan-Fang Li, and Shonali Krishnaswamy. A Rigorous Characterization of Classification Performance – A Tale of Four Reasoners. In Ian Horrocks, Mikalai Yatskevich, and Ernesto Jimenez-Ruiz, editors, *OWL Reasoner Evaluation Workshop (ORE 2012)*, volume 858 of *CEUR Workshop Proceedings*, pages 88–99. CEUR-WS, 2012.

[56] Yevgeny Kazakov and Pavel Klinov. Experimenting with ELK Reasoner on Android. In *2nd International Workshop on OWL Reasoner Evaluation (ORE-2013)*, pages 68–74, 2013.

[57] Yevgeny Kazakov, Markus Krötzsch, and František Simančík. The incredible ELK. *Journal of automated reasoning*, 53(1):1–61, 2014.

[58] Taehun Kim, Insuk Park, Soon J Hyun, and Dongman Lee. MiRE4OWL: Mobile Rule Engine for OWL. In *2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, pages 317–322. IEEE, 2010.

[59] Holger Knublauch, Ray W. Fergerson, Natalya Fridman Noy, and Mark Alan Musen. The Protégé OWL plugin: An open development environment for Semantic Web applications. In *International Semantic Web Conference*, pages 229–243. Springer, 2004.

[60] KNX association. KNX. Open standard, KNX association, 2002. `https://knx.org`.

[61] Patrick Koopmann, Marcus Hähnel, and Anni-Yasmin Turhan. Energy-Efficiency of OWL Reasoners – Frequency Matters. In *Joint International Semantic Technology Conference*, pages 86–101. Springer, 2017.

[62] Jean-Baptiste Lamy. Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies. *Artificial intelligence in medicine*, 80:11–28, 2017.

[63] Peter John Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, January 1964.

[64] Laozi, Gia-Fu Feng, and Jane English. *Tao Te Ching*. Random House - Vintage Books, August 1972.

[65] Mikhail K. Levin and Lindsay G. Cowell. owlcpp: a C++ library for working with OWL ontologies. *Journal of Biomedical Semantics*, 6(1):35, Sep 2015.

[66] Lei Li and Ian Horrocks. A Software Framework for Matchmaking Based on Semantic Web Technology. *International Journal of Electronic Commerce*, 8(4), 2004.

[67] Joshua Lieberman, Raj Singh, and Chris Goad. W3C Geospatial Vocabulary. W3C Incubator Group Report, W3C Geospatial Incubator Group (GeoXG), Oct 2007.

[68] Thorsten Liebig, Marko Luther, and Olaf Noppens. OWLlink: Structural Specification. Member Submission, W3C, July 2010. `https://www.w3.org/submissions/owllink-structural-specification`.

[69] Thorsten Liebig, Marko Luther, Olaf Noppens, and Michael Wessel. Owllink. *Semantic Web*, 2(1):23–32, January 2011.

[70] Phillip Lord and Jennifer D. Warrender. Horned-owl: Building ontologies at big data scale. In *Proceedings of the International Conference on Biomedical Ontologies 2021 (ICBO 2021), Bozen-Bolzano, Italy, 16-18 September, 2021*, volume 3073 of *CEUR Workshop Proceedings*, pages 134–136. CEUR-WS.org, 2021.

[71] Giuseppe Loseto, Ivano Bilenchi, Filippo Gramegna, Davide Loconte, Floriano Scioscia, and Michele Ruta. Tiny-ME Wasm: Description

Logics Reasoning in Your Browser. In Casteleyn *et al.*, editor, *Current Trends in Web Engineering*, pages 114–126, Cham, 2024. Springer Nature Switzerland.

[72] Giuseppe Loseto, Floriano Scioscia, Michele Ruta, Filippo Gramegna, and Ivano Bilenchi. Semantic-based adaptation of quality of experience in web multimedia streams. In *38th ACM/SIGAPP Symposium On Applied Computing (SAC 2023)*, pages 1821–1830. ACM, ACM Press, March 2023.

[73] Giuseppe Loseto, Floriano Scioscia, Michele Ruta, Filippo Gramegna, Saverio Ieva, Corrado Fasciano, Ivano Bilenchi, and Davide Loconte. Osmotic Cloud-Edge Intelligence for IoT-based Cyber-Physical Systems. *Sensors*, 22(6):2166, 2022.

[74] Giuseppe Loseto, Floriano Scioscia, Michele Ruta, Filippo Gramegna, Saverio Ieva, Corrado Fasciano, Ivano Bilenchi, Davide Loconte, and Eugenio Di Sciascio. A Cloud-Edge Artificial Intelligence Framework for Sensor Networks. In *9th IEEE International Workshop on Advances in Sensors and Interfaces (IWASI 2023)*, pages 149–154, 2023.

[75] Brian McBride. Jena: A Semantic Web toolkit. *IEEE Internet computing*, 6(6):55–59, 2002.

[76] Martín O. Moguillansky, Renata Wassermann, and Marcelo A. Falappa. An argumentation machinery to reason over inconsistent ontologies. In Guillermo R Simari Angel Kuri-Morales, editor, *Advances in Artificial Intelligence–IBERAMIA 2010*, pages 100–109, Berlin, Germany, 2010. Springer.

[77] Boris Motik, Bijan Parsia, and Peter Patel-Schneider. OWL 2 Web Ontology Language XML Serialization (Second Edition). Recommendation, W3C, December 2012. `https://www.w3.org/TR/owl2-xml-serialization`.

[78] Mark Alan Musen. The Protégé project: a look back and a look forward. *AI matters*, 1(4):4–12, 2015.

[79] Bojan Najdenov, Goran Petkovski, Milos Jovanovik, Riste Stojanov, and Dimitar Trajanov. Automated linked data generation from the transport administration domain. In *2015 23rd Telecommunications Forum Telfor (TELFOR)*, pages 827–830, 2015.

[80] Tu Ngoc Nguyen and Wolf Siberski. SLUBM: An Extended LUBM Benchmark for Stream Reasoning. In *2nd International Workshop on Ordering and Reasoning, in the 12th International Semantic Web Conference (ISWC 2013)*, volume 1059 of *CEUR Workshop Proceedings*, pages 43–54, 2013.

[81] Olaf Noppens, Marko Luther, and Thorsten Liebig. The OWLlink API: Teaching OWL Components a Common Protocol. In *Proceedings of the 7th International Workshop on OWL: Experiences and Directions (OWLED 2010)*, pages 13.1–13.4, 2010.

[82] Tomoki Okuro, Yumiko Nakayama, Yoshitada Takeshima, Yusuke Kondo, Nobuya Tachimori, Makoto Yoshida, Hiromu Yoshihara, Hirohiko Suwa, and Keiichi Yasumoto. Vehicle Detection and Classification using Vibration Sensor and Machine Learning. In *2022 18th International Conference on Intelligent Environments (IE)*, pages 1–8, 2022.

[83] Massimo Paolucci, Takahiro Kawamura, Terry R Payne, and Katia Sycara. Semantic Matching of Web Services Capabilities. In *International Semantic Web Conference*, pages 333–347. Springer, 2002.

[84] Bijan Parsia, Nicolas Matentzoglu, Rafael Gonçalves, Birte Glimm, and Andreas Steigmiller. The OWL reasoner evaluation (ORE) 2015 competition report. *Journal of Automated Reasoning*, 59(4):455–482, 2017.

[85] Bijan Parsia, Boris Motik, and Peter Patel-Schneider. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition). Recommendation, W3C, December 2012. `http://www.w3.org/TR/owl2-syntax/`.

[86] Evan W. Patton and Deborah L. McGuinness. A power consumption benchmark for reasoners on mobile devices. In *International Semantic Web Conference*, pages 409–424. Springer, 2014.

[87] Charith Perera, Arkady Zaslavsky, Chi Harold Liu, Michael Compton, Peter Christen, and Dimitrios Georgakopoulos. Sensor Search Techniques for Sensing as a Service Architecture for the Internet of Things. *Sensors Journal, IEEE*, 14(2):406–420, 2014.

[88] Andrea Pompigna and Raffaele Mauro. Smart roads: A state of the art of highways innovations in the smart age. *Engineering Science and Technology, an International Journal*, 25:100986, 2022.

[89] Filippo Giammaria Praticò, Gaetano Bosurgi, Dario Bruneo, Salvatore Cafiso, Fabrizio De Vita, Alessandro Di Graziano, Rosario Fedele, Orazio Pellegrino, and Giuseppe Sollazzo. Innovative smart road management systems in the urban context: Integrating smart sensors and miniaturized sensing systems. *Structural Control and Health Monitoring*, 29(10):e3044, 2022.

[90] Eric Prud'hommeaux and Gavin Carothers. RDF 1.1 Turtle. Recommendation, W3C, February 2014. `https://www.w3.org/TR/turtle/`.

[91] Andreas Rossberg. WebAssembly Specification Release 2.0 (Draft 2023-04-08). `https://webassembly.github.io/spec/core/`. Accessed: 2023-04-14.

[92] Michele Ruta, Tommaso Di Noia, Eugenio Di Sciascio, Giacomo Piscitelli, and Floriano Scioscia. Semantic-based mobile registry for dynamic RFID-based logistics support. In *10th International Conference on Electronic Commerce, ICEC 08*, pages 1–9. ACM Press, 2008.

[93] Michele Ruta, Eugenio Di Sciascio, and Floriano Scioscia. Concept Abduction and Contraction in Semantic-based P2P Environments. *Web Intelligence and Agent Systems*, 9(3):179–207, 2011.

[94] Michele Ruta, Floriano Scioscia, Ivano Bilenchi, Filippo Gramegna, Giuseppe Loseto, Saverio Ieva, and Agnese Pinto. A multiplatform

reasoning engine for the Semantic Web of Everything. *Journal of Web Semantics*, 73:100709, 2022.

[95] Michele Ruta, Floriano Scioscia, and Eugenio Di Sciascio. Enabling the Semantic Web of Things: framework and architecture. In *Sixth IEEE International Conference on Semantic Computing (ICSC 2012)*, pages 345–347, 2012.

[96] Michele Ruta, Floriano Scioscia, Eugenio Di Sciascio, and Ivano 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)*, volume 10161 of *Lecture Notes in Computer Science*, pages 141–152. W3C, Springer, Nov 2016.

[97] Michele Ruta, Floriano Scioscia, Eugenio Di Sciascio, and Domenico Rotondi. Ubiquitous Knowledge Bases for the Semantic Web of Things. In *First Internet of Things International Forum*, November 2011.

[98] Michele Ruta, Floriano Scioscia, Filippo Gramegna, Ivano Bilenchi, and Eugenio Di Sciascio. Mini-ME Swift: the first OWL reasoner for iOS. In *16th Extended Semantic Web Conference (ESWC 2019)*, pages 298–313. Springer, 2019.

[99] Michele Ruta, Floriano Scioscia, Giuseppe Loseto, Filippo Gramegna, Saverio Ieva, Agnese Pinto, and Eugenio Di Sciascio. Social Internet of Things for Domotics: a Knowledge-based Approach over LDP-CoAP. *Semantic Web Journal*, 9(6):781–802, 2018.

[100] Michele Ruta, Floriano Scioscia, Giuseppe Loseto, Agnese Pinto, and Eugenio Di Sciascio. Machine learning in the Internet of Things: A semantic-enhanced approach. *Semantic Web*, 10(1):183–204, 2019.

[101] Michele Ruta, Floriano Scioscia, Giuseppe Loseto, Agnese Pinto, and Eugenio Di Sciascio. Machine learning in the Internet of Things: A semantic-enhanced approach. *Semantic Web*, 10(1):183–204, 2019.

[102] Michele Ruta, Floriano Scioscia, Agnese Pinto, Filippo Gramegna, Saverio Ieva, Giuseppe Loseto, and Eugenio Di Sciascio. CoAP-based

collaborative sensor networks in the Semantic Web of Things. *Journal of Ambient Intelligence and Humanized Computing*, 10(7):2545–2562, jul 2019.

[103] Guus Schreiber and Fabien Gandon. RDF 1.1 XML syntax. Recommendation, W3C, February 2014. `http://www.w3.org/TR/rdf-syntax-grammar/`.

[104] Floriano Scioscia, Ivano Bilenchi, Michele Ruta, Filippo Gramegna, and Davide Loconte. A multiplatform energy-aware OWL reasoner benchmarking framework. *Journal of Web Semantics*, 72:100694, 2022.

[105] Floriano Scioscia, Giuseppe Loseto, Arnaldo Tomasino, Ivano Bilenchi, Filippo Gramegna, Saverio Ieva, Agnese Pinto, Eugenio Di Sciascio, and Michele Ruta. Embedded reasoning for uav operations: towards real-time efficiency and trustworthy autonomy. In *9th Italian Conference on ICT for Smart Cities And Communities (I-CiTies 2023)*, sep 2023.

[106] Floriano Scioscia and Michele Ruta. Building a Semantic Web of Things: issues and perspectives in information compression. In *Proceedings of the 3rd IEEE International Conference on Semantic Computing*, pages 589–594. IEEE Computer Society, 2009.

[107] Floriano Scioscia, Michele Ruta, Giuseppe Loseto, Filippo Gramegna, Saverio Ieva, Agnese Pinto, and Eugenio Di Sciascio. Mini-ME matchmaker and reasoner for the Semantic Web of Things. In *Innovations, Developments, and Applications of Semantic Web and Information Systems*, pages 262–294. IGI Global, 2018.

[108] Fred Shaffer and Jay P. Ginsberg. An overview of heart rate variability metrics and norms. *Frontiers in public health*, page 258, 2017.

[109] Hazim Shakhatreh, Ahmad H. Sawalmeh, Ala Al-Fuqaha, Zuochao Dou, Eyad Almaita, Issa Khalil, Noor Shamsiah Othman, Abdallah Khreishah, and Mohsen Guizani. Unmanned Aerial Vehicles (UAVs): A Survey on Civil Applications and Key Research Challenges. *IEEE Access*, 7:48572–48634, 2019.

[110] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3:637–646, 2016.

[111] Tasnuba Siddiqui and Bashir I. Morshed. Severity classification of chronic obstructive pulmonary disease and asthma with heart rate and SpO2 sensors. In *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 2929–2932. IEEE, 2018.

[112] Alex Sinner and Thomas Kleemann. Krhyper–in your pocket. In *International Conference on Automated Deduction*, pages 452–457. Springer, 2005.

[113] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.

[114] Steffen Staab and Rudi Studer. *Handbook on ontologies*. Springer Science & Business Media, 2010.

[115] Andreas Steigmiller, Thorsten Liebig, and Birte Glimm. Konclude: system description. *Journal of Web Semantics*, 27:78–85, 2014.

[116] Vitalijs Stepanovs. BrandT: Browser Hosted OWL Reasoner. Technical report, University of Mancherster, May 2011.

[117] Fredrik Sundbom, Christer Janson, Mirjam Ljunggren, and Eva Lindberg. Asthma and asthma-related comorbidity: effects on nocturnal oxygen saturation. *Journal of Clinical Sleep Medicine*, 18(11):2635–2641, 2022.

[118] Wei Tai, John Keeney, and Declan O'Sullivan. Resource-constrained reasoning using a reasoner composition approach. *Semantic Web*, 6(1):35–59, 2015.

[119] Antero Taivalsaari and Tommi Mikkonen. The Web as a Software Platform: Ten Years Later. In *13th International Conference on Web Systems and Technologies (WEBIST'17)*, pages 41–50, 2017.

[120] Gunnar 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)*, pages 540–550. ACM, 1994.

[121] Mehdi Terdjimi, Lionel Médini, and Michael Mrissa. HyLAR+ improving hybrid location-agnostic reasoning with incremental rule-based update. In *Proceedings of the 25th International Conference Companion on World Wide Web*, pages 259–262, 2016.

[122] The W3C SPARQL Working Group. SPARQL 1.1 Overview. Recommendation, W3C, March 2013. `https://www.w3.org/TR/sparql11-overview/`.

[123] Edward Thomas, Jeff Z. Pan, and Yuan Ren. TrOWL: Tractable OWL 2 reasoning infrastructure. In *Extended Semantic Web Conference (ESWC)*, pages 431–435, Berlin, Germany, 2010. Springer.

[124] Dmitry Tsarkov and Ian Horrocks. FaCT++ description logic reasoner: System description. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 292–297, Berlin, Germany, 2006. Springer.

[125] Edgaras Valincius, Hai H Nguyen, and Jeff Z Pan. A Power Consumption Benchmark Framework for Ontology Reasoning on Android Devices. In *OWL Reasoner Evaluation (ORE) Workshop*, pages 80–86, 2015.

[126] William Van Woensel and Syed Sibte Raza Abidi. Optimizing Semantic Reasoning on Memory-Constrained Platforms Using the RETE Algorithm. In *Extended Semantic Web Conference (ESWC)*, pages 682–696. Springer, 2018.

[127] William Van Woensel and Syed Sibte Raza Abidi. Benchmarking semantic reasoning on mobile platforms: Towards optimization using OWL2 RL. *Semantic Web*, 10(4):637–663, 2019.

[128] William Van Woensel and Syed Sibte Raza Abidi. Benchmarking semantic reasoning on mobile platforms: Towards optimization using OWL2 RL. *Semantic Web*, 10(4):637–663, 2019.

[129] William Van Woensel, Newres Al Haider, Ahmad Ahmad, and Syed SR Abidi. A cross-platform benchmark framework for mobile Semantic Web reasoning engines. In *International Semantic Web Conference*, pages 389–408. Springer, 2014.

[130] William Van Woensel, Floriano Scioscia, Giuseppe Loseto, Oshani Seneviratne, Evan Patton, Samina Abidi, and Lalana Kagal. Explainable clinical decision support: towards patient-facing explanations for education and long-term behavior change. In *International Conference on Artificial Intelligence in Medicine*, pages 57–62. Springer, 2022.

[131] Ruben Verborgh and Jos De Roo. Drawing conclusions from Linked Data on the Web: the EYE reasoner. *IEEE Software*, 32(3):23–27, 2015.

[132] Silviu Vert, Bogdan Dragulescu, and Radu Vasiu. LOD4AR: Exploring Linked Open Data with a Mobile Augmented Reality Web Application. In *ISWC (Posters & Demos)*, pages 185–188, 2014.

[133] Zekun Wang, Juan Cui, Tingshan Liu, Shanming Bai, Congcong Hao, Yongqiu Zheng, and Chenyang Xue. Composited pressure-velocity sensor based on sandwich-like triboelectric nanogenerator for smart traffic monitoring. *IEEE Sensors Journal*, 2023.

[134] Charlotte Weil, Simon Elias Bibri, Régis Longchamp, François Golay, and Alexandre Alahi. A Systemic Review of Urban Digital Twin Challenges, and Perspectives for Sustainable Smart Cities. *Sustainable Cities and Society*, page 104862, 2023.

[135] Patricia Whetzel, Natasha Noy, Nigam Shah, Paul Alexander, Csongor Nyulas, Tania Tudorache, and Mark Musen. Bioportal: Enhanced functionality via new web services from the national center for biomedical ontology to access and use ontologies in software applications. *Nucleic acids research*, 39:W541–5, 06 2011.

[136] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[137] Zhoujing Ye, Guannan Yan, Ya Wei, Bin Zhou, Ning Li, Shihui Shen, and Linbing Wang. Real-Time and Efficient Traffic Information Acquisition via Pavement Vibration IoT Monitoring System. *Sensors*, 21(8), 2021.

[138] Roberto Yus and Primal Pappachan. Are Apps Going Semantic? A Systematic Review of Semantic Mobile Applications. In *1st International Workshop on Mobile Deployment of Semantic Technologies*, volume 1506 of *CEUR Workshop Proceedings*, pages 2–13, 2015.

[139] A. A. Zaidan, B. B. Zaidan, Muzammil Hussain, Ahmed Haiqi, M. L. Mat Kiah, and Mohamed Abdulnabi. Multi-criteria analysis for OS-EMR software selection problem: A comparative study. *Decision Support Systems*, 78:15–27, 2015.

List of publications

Journal articles

1. Floriano Scioscia, Ivano Bilenchi, Michele Ruta, Filippo Gramegna, and Davide Loconte. A multiplatform energy-aware OWL reasoner benchmarking framework. *Journal of Web Semantics*, 72:100694, 2022

2. Michele Ruta, Floriano Scioscia, Ivano Bilenchi, Filippo Gramegna, Giuseppe Loseto, Saverio Ieva, and Agnese Pinto. A multiplatform reasoning engine for the Semantic Web of Everything. *Journal of Web Semantics*, 73:100709, 2022

3. Giuseppe Loseto, Floriano Scioscia, Michele Ruta, Filippo Gramegna, Saverio Ieva, Corrado Fasciano, Ivano Bilenchi, and Davide Loconte. Osmotic Cloud-Edge Intelligence for IoT-based Cyber-Physical Systems. *Sensors*, 22(6):2166, 2022

Peer-reviewed conference papers

1. Giuseppe Loseto, Ivano Bilenchi, Filippo Gramegna, Davide Loconte, Floriano Scioscia, and Michele Ruta. Tiny-ME Wasm: Description Logics Reasoning in Your Browser. In Casteleyn *et al.*, editor, *Current Trends in Web Engineering*, pages 114–126, Cham, 2024. Springer Nature Switzerland

2. Giuseppe Loseto, Floriano Scioscia, Michele Ruta, Filippo Gramegna, Saverio Ieva, Corrado Fasciano, Ivano Bilenchi, Davide Loconte, and Eugenio Di Sciascio. A Cloud-Edge Artificial Intelligence Framework

for Sensor Networks. In *9th IEEE International Workshop on Advances in Sensors and Interfaces (IWASI 2023)*, pages 149–154, 2023

3. Giuseppe Loseto, Floriano Scioscia, Michele Ruta, Filippo Gramegna, and Ivano Bilenchi. Semantic-based adaptation of quality of experience in web multimedia streams. In *38th ACM/SIGAPP Symposium On Applied Computing (SAC 2023)*, pages 1821–1830. ACM, ACM Press, March 2023

4. Floriano Scioscia, Giuseppe Loseto, Arnaldo Tomasino, Ivano Bilenchi, Filippo Gramegna, Saverio Ieva, Agnese Pinto, Eugenio Di Sciascio, and Michele Ruta. Embedded reasoning for uav operations: towards real-time efficiency and trustworthy autonomy. In *9th Italian Conference on ICT for Smart Cities And Communities (I-CiTies 2023)*, sep 2023

5. Ivano Bilenchi, Floriano Scioscia, and Michele Ruta. Cowl: A Lightweight OWL Library for the Semantic Web of Everything. In Agapito *et al.*, editor, *Current Trends in Web Engineering. ICWE 2022.*, pages 100–112, Cham, 2023. Springer

6. Ivano Bilenchi, Arnaldo Tomasino, Filippo Gramegna, Saverio Ieva, Agnese Pinto, Giuseppe Loseto, Floriano Scioscia, and Michele Ruta. Knowledge Representation and Reasoning for Unmanned Aerial Vehicle Intelligence. In *7th Italian Workshop on Embedded Systems (IWES 2022)*, 2022

7. Filippo Gramegna, Arnaldo Tomasino, Saverio Ieva, Ivano Bilenchi, Agnese Pinto, Giuseppe Loseto, Floriano Scioscia, and Michele Ruta. RideMATCHain: a Semantic-enhanced Blockchain Marketplace for Ridesharing. In *8th Italian Conference on ICT for Smart Cities And Communities (I-CiTies 2022)*, 2022