Optimizing and Benchmarking OWL2 RL for Semantic Reasoning on Mobile Platforms

William Van Woensel[a,*] and Syed Sibte Raza Abidi[a]
[a]*Faculty of Computer Science, Dalhousie University, 6050 University Ave, Halifax, NS B3H 1W5, Nova Scotia, Canada*

Abstract. The Semantic Web has grown immensely over the last decade, and mobile hardware has advanced to a point where mobile apps may consume this Web of Data. This has been exemplified in domains such as mobile context-awareness, m-Health, m-Tourism and augmented reality. However, recent work shows that the performance of ontology-based reasoning, an essential Semantic Web building block, still leaves much to be desired on mobile platforms. Applying OWL2 RL to realize such mobile reasoning is a promising solution, since it trades expressivity for scalability, and its rule-based axiomatization easily allows applying axiom subsets to improve performance. At any rate, considering the current performance issues, developers should be able to benchmark reasoners on mobile platforms, using different process flows, reasoning tasks, and datasets. To that end, we developed a mobile benchmark framework called MobiBench. In an effort to optimize mobile ontology-based reasoning, we further propose selections of OWL2 RL rule subsets based on logical equivalence, purpose and reference, and domain relevance. Using MobiBench, we benchmark multiple OWL2 RL-enabled rule engines and OWL reasoners on a mobile platform. Results show drastic performance improvements by applying OWL2 RL rule subsets, allowing for performant reasoning for small datasets on mobile systems.

Keywords: mobile computing; OWL2 RL; rule-based reasoning; OWL reasoning; reasoning optimization

1. Introduction

As illustrated by the Linking Open Data cloud [16], the Semantic Web currently includes over 1000 huge datasets from diverse domains such as government, music, geography, social web and life sciences. Lightweight structured data is being increasingly embedded in websites as well [78], motivated by search engines' need for machine-readable web content. Collectively, these advancements are giving rise to a multi-segmented, interlinked and machine-readable Web, ready for consumption by software agents. In particular, advances in mobile hardware and connectivity have enabled the mobile consumption of semantic data, e.g., to collect context- [57], [64] and location-related data [7], [55], achieve augmented reality [47], [63], perform recommendations [65], access linked biomedical data (m-Health) [45] and enable mobile tourism [30].

Regardless of the computing platform, an essential Semantic Web pillar is reasoning, which infers useful new information based on the semantics of ontology constructs, or deterministic, domain-specific if-then rules. However, many works have noted that OWL2 DL reasoning is too complex and resource-intensive to achieve scalability on mobile and embedded systems, which was confirmed by recent empirical work [28], [62]. In light of this issue, the state of the art has presented a number of semantic reasoners tailored to resource-constrained systems [1], [31], [41], [53]. Most of these approaches focus on rule-based OWL axiomatizations, such as custom entailment rulesets [1], [31] or an OWL2 RL ruleset (i.e., axiomatizing the OWL2 RL semantics) [41], [53]. Indeed, by utilizing the OWL2 RL W3C recommendation [13], any rule engine can implement a subset of OWL2 reasoning that allows for scalability without sacrificing too much expressive power. Furthermore, related work shows that subsets of rule-based axioms can be easily selected to e.g., adjust reasoning complexity to the ap-

[*] Corresponding author. E-mail: william.van.woensel@gmail.com.

plication scenario [53], or avoid resource-heavy inferences [9], [49]. In addition, OWL2 RL rules can be easily included into other rulesets to enhance the rule-based reasoning task (e.g., service matching) with ontology-based reasoning. In doing so, a single rule engine can perform semantically-enhanced reasoning without additional components (e.g., separate OWL reasoners). Because of these reasons, we argue that OWL2 RL is a promising solution for ontology-based reasoning on mobile, resource-constrained devices.

Nonetheless, it seems that mobile hardware will remain relatively limited compared to PC or server hardware, leaving mobile reasoning performance heavily dependent on dataset scale and application requirements [28], [62], at least in the foreseeable future. As such, there is a clear need for benchmarking reasoning performance in particular mobile application scenarios, including reasoning task (e.g., ontology or rule-based reasoning, service matching), process flow (e.g., frequent vs. incremental reasoning) and custom rule- and datasets. Based on the results, developers can make informed decisions about deploying semantic reasoning; with poor performance e.g., pointing to a distributed solution [2], [61]. Currently, systematic mobile benchmarking is impeded by the range of custom rule- and data formats and semantic rule standards (e.g., RuleML, SWRL, SPIN); as well as the fragmented mobile market, including OS's such as Android, iOS, Windows Phone, BlackBerry OS and Symbian.

We pursue multiple objectives: (a) optimize semantic reasoning on mobile platforms by selecting OWL2 RL subsets; (b) develop a mobile reasoning benchmark framework (called MobiBench); (c) perform multiple mobile reasoning benchmarks; and by executing these benchmarks, (d) study the usefulness of mobile semantic reasoning.

With respect to our objectives (a-d), this paper presents the following 4 contributions:

1) **A selection of OWL2 RL subsets**, featuring 3 selections to be applied in conjunction or in isolation:

- Equivalent OWL2 RL rule subset: this selection leaves out logically equivalent rules; i.e., rules of which the results are covered by other rules. Also, by introducing extra axioms, possibly combined with new, more general rules, multiple existing rules can be left out. Some rules may also be considered redundant at the instance level, inferring new schema elements but not contributing to new instances.

- Purpose and reference-based subsets: by dividing rule subsets via their purpose and referenced data, smaller rulesets can be applied in certain runtime scenarios. In particular, OWL2 RL rules perform either *inference* or *consistency-checking* (purpose), and refer to *instances* and *schema* or only *schema* elements (reference). Related work often separates instance- (ABox) from schema-based (TBox) reasoning [5], [17], [25], [40], [41]. Further, rules that will not yield inferences over the ontology can be left out as well, by applying a separate pre-processing step [53].

- Removal of inefficient rules: this selection leaves out rules with a large performance impact. While this will clearly result in missing certain OWL2 RL inferences, their large overhead implies that developers should at least be allowed to weigh their utility vs. computational cost.

2) **A mobile reasoning benchmark framework (MobiBench)** to evaluate the performance of reasoning on mobile platforms (reasoning times, memory usage), in specific scenarios and using standards-based rule- and datasets. To that end, the framework includes a *Semantic Web layer*, which supplies a uniform, standards-based interface across reasoning engines; a *Selection Service*, to automatically apply the selections from (1); and a *Pre-processing Service*, to pre-process a ruleset and ontology for a particular purpose (e.g., to support n-ary rules). Developers run benchmarks by invoking the API or using the built-in automation support. Analysis tools convert reasoning times and memory dumps into summary CSV files.

Key features of MobiBench include extensibility and its cross-platform nature, allowing benchmarks to be applied across multiple platforms. For this purpose, we implemented MobiBench in JavaScript (JS), and use Apache Cordova [67] for deployment on mobile platforms; and the JDK8 Nashorn JavaScript engine [73] for PC platforms (currently, this version is used mostly for testing). Currently, MobiBench is deployed on Android (using Cordova) and PC (using JDK 8).

3) **Mobile benchmarks**, measuring the performance of two rule engines, namely the Android-based AndroJena and the JS-based RDFStore-JS. Multiple reasoning tasks are benchmarked, namely materializing ontology inferences and performing semantically-enhanced service matching. Further, we benchmark each OWL2 RL subset selection to measure its performance impact. Three OWL2 DL reasoners (HermiT, JFact and Pellet) are also benchmarked. We utilize the OWL2 RL Benchmark Corpus and the OWL-S Service Retrieval Test Collection [32] for benchmarking.

4) **A study of the usefulness of OWL2 RL for semantic reasoning on mobile platforms**. The above contributions implement, optimize and apply OWL2

RL rules in practice, both 1) stand-alone, to materialize ontology inferences; and 2) combined with a service matching ruleset, to enhance service matching with ontology reasoning. Service matching is a useful task in mobile settings, as it enables mobile apps to locate relevant services in a smart environment [58]. Aside from performance, we also look at the impact of ontology reasoning on service matching results. Thus, our work contributes to studying the feasibility as well as utility of OWL2 RL on mobile platforms.

This paper is built on previous work, which presented a clinical benchmark [60] and an initial version of the Mobile Benchmark Framework [59], which only supplied an API and restricted benchmarking to rule-based (non-OWL) reasoning. As such, it did not attempt optimizations or applications of OWL2 RL.

The paper is structured as follows. In Section 2, we shortly discuss the OWL2 RL profile and its implementation. Section 3 explains our selection of OWL2 RL rule subsets. Section 4 presents the architecture and main components of MobiBench, and Section 5 discusses how mobile developers can utilize MobiBench. Section 6 elaborates on the benchmarks we performed using MobiBench. We discuss related work in Section 7, and end with conclusions and future work in Section 8.

2. OWL2 RL Realization

This section shortly discusses the different OWL2 profiles (Section 2.1) and our practical implementation of OWL2 RL (Section 2.2).

2.1. OWL2 Profiles

The OWL2 Web Ontology Language Profiles document [13] introduces multiple OWL2 profiles, including OWL2 EL, OWL2 QL and OWL2 RL. By restricting ontology syntax and reducing expressivity, these profiles can more efficiently handle specific application scenarios. OWL2 EL is designed to deal with ontologies containing large amounts of classes and properties, whereas OWL2 QL is optimized for querying large amounts of instance data. The OWL2 RL profile is aimed at balancing expressivity with reasoning scalability, and presents a partial, rule-based axiomatization of OWL2 RDF-Based Semantics. Using OWL2 RL, reasoning systems can be implemented using standard rule engines. The W3C specification [13] presents the OWL2 RL axiomatization as a set of universally quantified, first-order implications over a ternary predicate T, which stands for a generalization of RDF triples. In addition to regular inference rules, the ruleset includes rules that are always applicable (i.e., without antecedent), and consistency-checking rules (i.e., with consequent *false*). Below, we exemplify each type of rules (namespaces omitted for brevity). Code 1 shows a “regular” inference rule that types resources based on the *subClassOf* construct:

$$T(?c_1, subClassOf, ?c_2), T(?x, type, ?c_1) \rightarrow T(?x, type, ?c_2)$$

Code 1. Rule classifying resources (*#cax-sco*).

The second type of rule lacks an antecedent and is thus always applicable. E.g., the rule in Code 2 indicates that each built-in OWL2 RL annotation property needs to have the *owl:AnnotationProperty* type:

$$T(ap, type, AnnotationProperty)$$

Code 2. Rule typing annotation properties (*#prp-ap*).

Thirdly, the consistency-checking rule in **Code 3** checks whether an instance of a restriction, indicating a maximum cardinality of 0 on a particular property, participates in said property. If so, the ontology is flagged as inconsistent.

$$T(?x, maxCardinality, 0), T(?x, onProperty, ?p),$$
$$T(?u, type, ?x), T(?u, ?p, ?y) \rightarrow false$$

Code 3. Rule based on maxCardinality restriction to check consistency (*#cls-maxc1*).

Below, we elaborate on our practical realization of OWL2 RL for arbitrary rule engines.

2.2. Practical Realization of OWL2 RL

To implement the OWL2 RL axiomatization for general-purpose rule engines, where no particular internal support can be assumed, three types of rules may pose problems: 1) rules that require internal datatype support; 2) rules that are always applicable; and 3) rules referring to lists of elements. Below, we present these issues and our solutions.

2.2.1. Rules requiring datatype support

The datatype inference rule *#dt-type2* (Code 4) requires literals with data values from a certain value space to be typed with the datatype of that value space (e.g., typing an integer “42” with *xsd:int*):

$$T(?lt, type, ?dt)$$

Code 4. Rule typing each literal with its corresponding datatype (*#dt-type2*).

Similarly, a second rule (*#dt-not-type*) flags an inconsistency when a literal is typed with the wrong datatype. Two other datatype rules (*#dt-eq* and *#dt-diff*) indicate equality and inequality of literals based on their values; which requires differentiating literals from URIs, to avoid these rules to fire for URI resources as well. These four rules require built-in support for RDF datatypes and literals, meaning they cannot be consistently implemented across arbitrary rule engines. Therefore, we chose to leave these rules out of our OWL2 RL ruleset. Related work, including DLEJena [40] and the SPIN [33] and OWLIM [9] OWL2 RL rulesets also do not include datatype rules.

2.2.2. Always-applicable rules

A number of OWL2 RL rules lack an antecedent, and are thus always applicable. One subset of these rules lack variables (e.g., specifying that *owl:Thing* has type *owl:Class*), and may thus be directly represented as axiomatic triples to accompany the OWL2 RL ruleset. A second subset comprises "quantified" variables in the consequent; e.g., stating that each annotation property has type *owl:AnnotationProperty* (Code 2). Likewise, these were implemented by axioms that properly type each annotation property (built-in for OWL2 [24]) and datatype property (supported by OWL2 RL [13]).

2.2.3. Rules referencing element lists

This set of rules includes so-called *n-ary* rules, which refer to a finite list of elements. A first subset **(L1)** of these rules enumerate (i.e., list one by one) restrictions on single list elements (*#eq-diff2, #eq-diff3, #prp-adp, #cax-adc, #cls-uni*). For instance, rule *#eq-diff2* flags an ontology inconsistency if two equivalent elements of an *owl:AllDifferent* construct are found.

In contrast, rules from the second subset **(L2)** include restrictions referring to all list elements (*#prp-spo2, #prp-key, #cls-int1*), and a third ruleset **(L3)** yields inferences for all list elements (*#cls-int2, #cls-oo, #scm-int, #scm-uni*). E.g., for **(L2)**, rule *#cls-int1* infers that y is an instance of an intersection in case it is typed by *each* intersection member class; regarding **(L3)**, for any union, rule *#scm-uni* (Code 6) infers that *each* member class is a subclass of that union.

To support rulesets **(L1)** and **(L3)**, we added two list-membership rules (Code 5) that recursively link each element to preceding list cells, eventually linking the first cell to all list elements:

$$T(?\,l, first, ?\,m) \rightarrow T(?\,l, hasMember, ?\,m) \qquad \textbf{a)}$$
$$T(?\,l_1, rest, ?\,l_2), T(?\,l_2, hasMember, ?\,m) \rightarrow T(?\,l_1, hasMember, ?\,m) \qquad \textbf{b)}$$

Code 5. Two rules for inferring list membership.

Using these rules, *#scm-uni* **(L3)** may be formulated as follows (Code 6):

$$T(?\,c, unionOf, ?\,l), T(?\,l, hasMember, ?\,cl) \rightarrow T(?\,cl, subClassOf, ?\,c)$$

Code 6. Rule inferring subclasses based on union membership (*#scm-uni*).

Since the supporting rules (Code 5) link all list elements to the first list cell (i.e., *?l*) using *hasMember* assertions, the rule yields inferences for all union member classes.

However, extra support is required for **(L2)**. For these kinds of n-ary rules, we supply three solutions, each with their own advantages and drawbacks:

(1) *Instantiate the rules* based on n-ary assertions found in the ontology. Per OWL2 RL rule, this approach generates a separate rule for each related n-ary assertion, by constructing a list of the found length and instantiating variables with concrete schema references. E.g., a property chain axiom P with constituent properties P_{1-3} will yield the following rule (Code 7):

$$T(?\,u1, P1, ?\,u2), T(?\,u2, P2, ?\,u3), T(?\,u3, P3, ?\,u4) \rightarrow T(?\,u1, P, ?\,u4)$$

Code 7. Instantiated rule supporting a specific property chain axiom (*#prp-spo2*).

Some related works apply this approach (a.k.a. "rule-templates") for any n-ary rule [42], or even all (applicable) OWL2 RL rules [5], [40], [41].

A drawback of this approach is that it requires pre-processing the ruleset for each ontology, and whenever it changes. Although our selections also include a pre-processing option (Section 3.2), this is only needed for optimization. Of course, the severity of this drawback depends on the frequency of ontology updates. Also, it yields an extra rule for each relevant assertion, potentially inflating the ruleset. On the other hand, instantiated rules contain less variables, and may also reduce the need for joins, as for *#prp-spo2* (see also [41]). Further, in case no related assertions are found, no rules will be added the ruleset. Future work includes studying the application of this approach to all rules (Section 8).

(2) *Normalize (or "binarize") the input ontology* to only contain binary versions of relevant n-ary assertions. E.g., an n-ary intersection can be converted to a set of binary intersections as follows (Code 8):

$$I = C_1 \cap C_2 \cap \ldots \cap C_n \equiv$$
$$I = C_1 \cap I_2 \wedge I_2 = C_2 \cap I_3 \wedge \ \ldots \wedge \ I_{n-1} = C_{n-1} \cap C_n$$

Code 8. Binary version of an n-ary intersection.

With the binary version of *#cls-int1* (Code 9):

$$T(?\,c, intersectionOf, ?\,x_1), T(?\,x_1, first, ?\,c_1), T(?\,x_1, rest, ?\,x_2), T(?\,x_2, first, ?\,c_2), T(?\,y, type, ?\,c_1), T(?\,y, type, ?\,c_2) \rightarrow ?\,T(?\,y, type, ?\,c)$$

Code 9. Binary version of rule *#cls-int1*.

This rule may be considered recursive, since it both references and infers the same kind of assertion (i.e., $T(?y, type, ?c)$). Applying this rule on a set of binary assertions $I, I_2, \dots, I_{n-1}$ (see Code 8) yields the following for any resource R, with R_t representing its set of all types (Code 10):

$$\begin{cases} \{C_{n-1}, C_n\} \subset R_t \rightarrow R_t = \mathrm{R}_t + I_{n-1} \\ \{C_{i-1}, I_i\} \subset R_t \rightarrow R_t = R_t + I_{i-1} \quad (\boldsymbol{n-1 \geq i \geq 1}) \\ \{C_1, I_2\} \subset R_t \rightarrow R_t = R_t + I \end{cases}$$

Code 10. Inferences when applying binary *#cls-int1*.

In doing so, the rule travels up the chain of binary intersections, until it finally infers type *I* for *R*.

It is not hard to see how this approach only works for recursive rules. Rule *#prp-key* is not a recursive rule, since it infers equivalence between resources but does not refer to such relations. So, this approach only works for rules *#prp-spo2* and *#cls-int1* from **(L2)**. Another drawback is that, similar to (1), it requires pre-processing for each ontology and its updates. In particular, each relevant n-ary assertion needs to be replaced by $n - 1$ binarized versions. Further, to support a complete, single n-ary inference, this solution generates a total of $n - 1$ inferences. While these are sound inferences, they may be considered to "crowd" (i.e., expand) the dataset.

(3) *Replace each rule* from **(L2)** by *3 auxiliary rules*. Bishop et al. [9] suggested this solution for OWLIM, based on a W3C note [46]. In this solution, a first auxiliary rule starts at the end of any list, and infers an intermediary assertion for the last element (cell *n*). Starting from the first inference, a second rule travels up the list structure by inferring the same kind of assertions for cells i ($n > i \geq 0$). In case the first cell is related to a relevant n-ary assertion (e.g., intersection, property chain or has-key), a third auxiliary rule generates the original, n-ary inference. See Bishop et al. [9] or our online documentation [56] for details.

A distinct advantage of this approach is that, in contrast to (1) and (2), it does not rely on pre-processing. However, each complete, single n-ary inference requires a total of *n+1* inferences, and these do not follow from OWL2 RL semantics (instead, they ensue from custom, auxiliary rules). As such, they can be considered to not only "crowd" but also "pollute" the dataset with unsound inferences. Bishop et al. [9] internally flag these inferences so they are skipped in query answering. Developers may want to support a similar mechanism when adopting this solution.

Based on all observations from Section 2, we collected an OWL2 RL ruleset implementation written in the SPARQL Inferencing Notation (SPIN), based on an initial ruleset created by Knublauch [33]. This initial ruleset relies on built-in Apache Jena functions to implement the rules from Section 2.2.3. Such built-in support cannot be assumed for arbitrary rule engines, which are targeted by our ruleset. Also, it does not specify axioms (Section 2.2.2). Our ruleset contains 69 rules and 13 supporting axioms, and can be found in Appendix A. This ruleset includes the two list-membership rules (Code 5) for n-ary rules from sets (L1) and (L3) (Section 2.2.3). To add support for a particular solution for (L2), our Web service needs to be contacted (Section 4.3) to pre-process the necessary rules or ontology, and/or add the rules (e.g., binary versions, auxiliary rules) to the ruleset. Note that our evaluation does not compare the performance of these n-ary rule solutions; this is considered future work.

In Section 3.4, we discuss options for checking conformance with OWL2 RL semantics.

3. OWL2 RL Optimization

This section discusses OWL2 RL ruleset selections, with the goal of optimizing ontology-based reasoning on mobile platforms. We consider three selections: leaving out redundant rules (Section 3.1), dividing the ruleset based on rule purpose and references (Section 3.2), and removing inefficient rules (Section 3.3).

For the purpose of these selections, we introduce the terms *owl2rl-schema-completeness* and *owl2rl-instance-completeness*, to indicate when a selection respectively derives all *schema inferences* and *instance inferences* covered by the OWL2 RL axiomatization. Although OWL2 RL reasoning infers all ABox inferences over OWL2 RL-compliant ontologies, it does not cover all TBox inferences dictated by the OWL 2 semantics [37], [41], hence our introduction of these specialized terms. Further, we discuss conformance with the OWL2 RL W3C specification (Section 3.4).

3.1. Equivalent OWL2 RL subset

As mentioned by the OWL2 RL specification [13], the presented ruleset is not minimal, as certain rules are implied by others. The stated goal of this redundancy is to make the semantic consequences of OWL2 constructs self-contained. Although this is appropriate from a conceptual standpoint, this redundancy is not useful when aiming to optimize reasoning.

Aside from rules that are entailed by other rules (Section 3.1.1), opportunities also exist to leave out specialized rules by introducing extra axioms (Section

3.1.2) or replacement by generalized rules (Section 3.1.3). Some inference rules may also be considered redundant at the instance level, since they do not contribute to inferring instances (Section 3.1.4).

3.1.1. Entailments between OWL2 RL rules

A first set of rules is entailed by *#cax-sco* (see Code 1), each time combined with a second inference rule. For instance, *#scm-uni* (see Code 6) indicates that each class in a union is a subclass of that union. Together, these two rules entail the *#cls-uni* rule (Code 11). This rule infers that each instance of a union member is an instance of the union itself:

$$T(?c, unionOf, ?x), T(?x, hasMember, ?cl),\\ T(?y, type, ?cl) \rightarrow T(?y, type, ?c)$$

Code 11. Rule that infers membership to OWL unions (*#cls-uni*).

Code 12 shows that the rule *#cls-uni*, for each instantiation of the input variables, is covered by *#scm-uni* + *#cax-sco*:

$$T(?c, unionOf, ?x), T(?x, hasMember, ?cl) \rightarrow\\ T(?cl, subClassOf, ?c) \quad \textbf{a)}$$
$$T(?cl, subClassOf, ?c), T(?y, type, ?cl) \rightarrow\\ T(?y, type, ?c) \quad \textbf{b)}$$

Code 12. Entailment of *#cls-uni* by *#scm-uni,#cax-sco*.

Applying *#scm-uni* on two premises from *#cls-uni* returns inference (a). Then, *#cax-sco* is applied on the remaining premise, together with (a). This yields the inference in (b), which equals the *#cls-uni* consequent. As such, this rule may be left out without losing expressivity. Similarly, it can be shown that rules *#cls-int2, #cax-eqc1 and #cax-eqc2* are entailed by *#cax-sco*, each time combined with a schema-based rule.

A second set of inference rules is entailed by the *#prp-spo1* rule, each time combined with rules indicating equivalence between *owl:equivalent[Class|Property]* and *rdfs:sub[Class|Property]Of*. Similar to *#cax-sco*, *#prp-spo1* (**Code 13**) infers that resources related via a sub property are also related via its super property:

$$T(?p_1, subPropertyOf, ?p_2), T(?x, ?p_1, ?y) \rightarrow T(?x, ?p_2, ?y)$$

Code 13. Rule that infers new resource relations *(#prp-spo1*).

E.g., the *#scm-eqp1* (**Code 14**) rule indicates that two equivalent properties are also sub properties:

$$T(?p_1, equivalentProperty, ?p_2)\\ \rightarrow T(?p_1, subPropertyOf, ?p_2), T(?p_2, subPropertyOf, ?p_1)$$

Code 14. Rule inferring sub properties (*#scm-eqp1*).

These two rules collectively entail the rule *#prp-eqp1* (Code 15). This rule infers that, for two equivalent properties, any resources related via the first property are also related via the second property:

$$T(?p_1, equivalentProperty, ?p_2), T(?x, ?p_1, ?y)\\ \rightarrow T(?x, ?p_2, ?y)$$

Code 15. Rule for property membership (*#prp-eqp1*).

This entailment is shown by Code 16:

$$T(?p_1, equivalentProperty, ?p_2) \rightarrow\\ T(?p_1, subPropertyOf, ?p_2) \quad \textbf{a)}$$
$$T(?p_1, subPropertyOf, ?p_2), T(?x, ?p_1, ?y) \rightarrow\\ T(?x, ?p_2, ?y) \quad \textbf{b)}$$

Code 16. Entailment of *#prp-eqp1* by *#scm-eqp1,#prp-spo1*.

By applying *#scm-eqp1* on the first premise from *#prp-eqp1*, the inference from (a) is returned. Applying *#prp-spo1* on this inference and the remaining premise yields (b), which equals the *#prp-eqp1* consequent. Therefore, this rule may be left out. Rule *#prp-eqp2* is similarly equivalent to these two rules as well.

Other rules are covered by single rule. The *#eq-trans* rule (Code 17) indicates the transitivity of *owl:sameAs*:

$$T(?x, sameAs, ?y), T(?y, sameAs, ?z) \rightarrow T(?x, sameAs, ?z)$$

Code 17. Rule indicating transitivity of owl:sameAs (*#eq-trans*).

This rule is entailed by *#eq-rep-o* (**Code 18**), which indicates that, for any triple, subject resources are related to any resource equivalent to the object:

$$T(?o, sameAs, ?o_2), T(?s, ?p, ?o) \rightarrow T(?s, ?p, ?o_2)$$

Code 18. Rule inferring new relations via owl:sameAs (*#eq-rep-o*).

By partially materializing the premise of *#eq-rep-o*, Code 19 shows how this rule entails *#eq-trans*:

$$T(?y, sameAs, ?z), T(?x, sameAs, ?y)\\ \rightarrow T(?x, sameAs, ?z)$$

Code 19. Entailment of *#eq-tra*ns by *#eq-rep-o*.

When executing the *#eq-rep-o* rule on suitable data, the *?p* variable is instantiated with *owl:sameAs*, thus covering each possible inference of *#eq-trans*.

Finally, we note that some rules could potentially be removed, depending on type assertions found in the dataset. Rules *#cls-maxqc4* & *#cls-svf2* support restrictions that apply to *owl:Thing*, and thus do not require objects to be typed with the restriction class (since each resource is implicitly already an *owl:Thing*). Related rules *#cls-maxqc3* & *#cls-svf2* support restrictions that apply to a particular class, and thus require related objects to be typed with the restriction class. Since *owl:Thing* is the supertype of each class (*#scm-cls* rule), and each instance is typed by its class's supertype (*#cax-sco* rule, Code 1), any instance will be typed as *owl:Thing*. Therefore, executing the second set of rules on restrictions relating to *owl:Thing* could produce the same inferences. However, *#cax-sco* requires each instance to be explicitly typed, which often is not the case in practice. Therefore, we opted to leave these rules in the ruleset.

We note that our online documentation [56] discusses all rule equivalences in detail. In total, this selection involved leaving out 7 redundant rules.

3.1.2. Extra supporting axiomatic triples

In other cases, extra axiomatic triples can be introduced to allow for entailment by existing rules. For instance, the rule *#eq-sym* (Code 20) explicitly encodes the symmetry of the *owl:sameAs* property:

$$T(?x, sameAs, ?y) \rightarrow T(?y, sameAs, ?x)$$

Code 20. Rule indicating owl:sameAs symmetry (*#eq-sym*).

By adding an axiom stating that *owl:sameAs* has type *owl:SymmetricProperty*, Code 21 shows that any inferences generated by the *#eq-sym* rule are covered by the *#prp-symp* rule:

$$T(?p, type, SymmetricProperty), T(?x, ?p, ?y) \rightarrow T(?y, ?p, ?x)$$
$$T(sameAs, type, SymmetricProperty)$$

Code 21. Rule implementing property symmetry *(#prp-symp)* and supporting axiom.

Similarly, *#prp-inv2* is entailed by *#prp-symp* with an extra axiom, together with the *#prp-inv1* rule.

Rules *#scm-spo* and *#scm-sco*, implementing the transitivity of *rdfs:subPropertyOf* and *rdfs:subClassOf*, respectively, are entailed by *#prp-trp* with supporting axioms (Code 22):

$$T(?p, type, TransitiveProperty), T(?x, ?p, ?y), T(?y, ?p, ?z) \rightarrow T(?x, ?p, ?z)$$
$$T(subPropertyOf, type, TransitiveProperty)$$
$$T(subClassOf, type, TransitiveProperty)$$

Code 22. Rule implementing transitivity *(#prp-trp)* and supporting axioms.

In doing so, 4 rules can be left out, at the expense of adding 4 new supporting axioms.

3.1.3. New generalized OWL2 RL rules

Opportunities also exist to generalize multiple rules into a single rule, combined with supporting axioms. We observe that rules *#eq-rep-p* (Code 23) and *#prp-spo1* (see Code 13) are structurally very similar:

$$T(?p, sameAs, ?p_2), T(?s, ?p, ?o) \rightarrow T(?s, ?p_2, ?o)$$

Code 23. Rule inferring new relations via owl:sameAs (*#eq-rep-p*).

Therefore, both rules can be generalized into a single rule, with accompanying axioms (Code 24):

$$T(?p_1, ?p, ?p_2), T(?p, type, SubLink), T(?s, ?p_1, ?o) \rightarrow T(?s, ?p_2, ?o)$$
$$sameAs\ \ type\ \ SubLink\ .$$
$$subPropertyOf\ \ type\ \ SubLink\ .$$

Code 24. Rule covering *#eq-rep-p* and *#prp-spo1* (*#prp-sl*) and supporting axioms.

In fact, several rules are structurally very similar, and may be pairwise generalized into a single rule with supporting axioms: rules *#scm-hv* and *#scm-svf2*; *#scm-avf1* and *#scm-svf1; #eq-diff2* and *#eq-diff3*; *#prp-npa1* and *#prp-npa2*; and *#cls-com* and *#cax-dw* (see [56] for details). In doing so, we left out 12 specialized rules, at the expense of adding 6 new general rules and 12 supporting axioms. After applying these selections, 52 rules remain and 16 axioms are added.

We note that these selections represent a best-effort in creating a *minimal* OWL2 RL-conformant rule subset, and do not necessarily optimize the ruleset for all types of systems. Although the total number of rules is reduced, specific rules are also being removed or replaced by more general rules; which could negatively impact performance. Our evaluation (Section 6) compares the effects of each subset selection.

3.1.4. Equivalence with instance-based rules

So-called “stand-alone” schema inferences, which extend the ontology but do not impact the set of instances, may also be considered redundant (at least, at the instance level). E.g., *#scm-dom1* (Code 25) infers that properties also have as domain the super types of their domains:

$$T(?p, domain, ?c_1), T(?c_1, subClassOf, ?c_2) \rightarrow T(?p, domain, ?c_2)$$

Code 25. Rule inferring super class domains (*#scm-dom1*).

Although this information may be a useful addition to the ontology, the new schema element will not result in new instance inferences. Code 26 shows that its resulting instance inferences are already covered by rules *#prp-dom* (a) and *#cax-sco* (b):

$$T(?s, ?p, ?o), T(?p, domain, ?c_1) \rightarrow T(?s, type, ?c_1) \quad \textbf{a)}$$
$$T(?s, type, ?c_1), T(?c_1, subClassOf, ?c_2) \rightarrow T(?s, type, ?c_2) \quad \textbf{b)}$$

Code 26. Two rules yielding same instance inferences as *#scm-dom1*.

Thus, any variable *?s* will already be typed with super classes of the property’s domain, regardless of the inferences generated by *#scm-dom1*. Similarly, rules *#scm-rng1*, *#scm-dom2* and *#scm-rng2* will not yield any new instances. By leaving out these 4 rules, this selection retains *owl2rl-instance-completeness* but clearly breaks *owl2rl-schema-completeness*.

3.2. Purpose- and reference-based subsets

Seeing how many (e.g., context-aware [41]) scenarios only involve adding or updating ABox (instance) statements at runtime, an option is to restrict TBox reasoning to design / startup time and whenever the ontology changes; and apply ABox reasoning when new instances are added. Reflecting this, most OWL2 RL reasoners focus on separating TBox from ABox reasoning [5], [17], [25], [40], [41]. Further, when data is generated by the system, it has a smaller likelihood of being inconsistent, thus reducing (or even removing) the need for continuous consistency checking.

Consequently, we divided our OWL2 RL ruleset into 2 major subsets; 1) *inference ruleset*, comprising inference rules (53 rules), and 2) *consistency-checking ruleset*, containing rules for checking consistency (18 rules[1]). The *inference ruleset* is further subdivided into 1.1) *instance ruleset*, consisting of rules inferring only instance assertions, while referring to both instance and schema elements (32 rules); and 1.2) *schema ruleset*, comprising rules only referencing schema elements (23 rules[1]). Since the consistency-checking ruleset only contains rules referring to both instance and schema elements, it cannot be further subdivided.

In this approach, *inference-schema* is applied on the ontology, initially and whenever the ontology changes, to materialize all schema inferences. When new instances are added, only *inference-instance* is applied on the instance assertions and materialized ontology. Below, we show that this process still produces a complete materialization.

Definition 1. We define S as the set of all schema assertions (i.e., TBox) and I the set of all instance assertions (i.e., ABox) with $S \cap I = \emptyset$, and $A = S \cup I$ the set of all assertions. We further define schema ruleset α and instance ruleset β as follows, with $IR = \alpha \cup \beta$ the set of all inference rules:

$$\begin{aligned} \alpha &= \{\, r \in IR \mid \forall c \in body(r), \\ &\qquad \forall a \in A : match(a, c) \rightarrow a \in S \,\} \\ \beta &= \{\, r \in IR \mid \forall i \in infer(r, A) : i \in I \,\} \end{aligned} \qquad \textbf{(1)}$$

Where $body(r)$ returns all clauses in the body of rule r, $match(a, c)$ returns true if assertion a matches a body clause c, and $infer(r, A)$ returns all inferences yielded by rule r on the set of assertions A. In other words, ruleset α includes rules for which each body clause is only matched by assertions from S, and ruleset β includes rules that only infer assertions from I. These conditions can be easily confirmed for our OWL2 RL rulesets [56]. $k^*(X)$ denotes the deductive closure of ruleset k on assertions X (i.e., returning X extended with any resulting inferences).

Theorem 1. The deductive closure of IR on the union of any ontology O ($O \subseteq S$) and dataset D ($D \subseteq I$) is equivalent to the deductive closure of β on the union of materialized ontology $\alpha^*(O)$ (i.e., including schema inferences) together with set of instances D:

$$(\alpha \cup \beta)^*(O \cup D) \equiv \beta^*(\alpha^*(O) \cup D) \qquad \textbf{(2)}$$

It is easy to see why this equivalence holds. Compared to the left operand, the set of assertions on which ruleset α is applied no longer includes inferences from ruleset β (since its deductive closure is now calculated separately), nor assertions from D. This does not affect the deductive closure of α, since α only matches assertions from S with $S \cap I = \emptyset$, and β only infers $i \in I$ (see Definition 1), whereas $D \subseteq I$. ∎

As indicated by our evaluation (Section 6), executing only the *inference-instance* ruleset has the potential to significantly improve performance. At the same time, the utility of separating these subsets depends on the frequency of ontology updates, since each update requires re-materializing the ontology. Although ontology changes are typically infrequent compared to instance data, this depends on the concrete scenario. Also, we note that related work often uses a separate OWL reasoner for materializing schema inferences [5], [17], [40]. Although this is a viable approach, we argue that this is not optimal for mobile platforms, as it requires deploying two resource-heavy components (i.e., an OWL reasoner and rule engine).

In the same vein, the *consistency-checking* ruleset needs to be applied on a dataset with all applicable inferences materialized, by a priori applying the *inference* ruleset. It can be similarly shown that applying only *consistency-checking* on such a dataset will not result in losing any consistency errors.

Finally, rules and axioms that are not referenced by the ontology may be left out as well, yielding a *domain-based* rule subset. For this purpose, we implemented a domain-based ruleset selection algorithm, which we elaborate in Section 4.2. Similar to before, its applicability depends on the frequency (and significance) of ontology updates; since this requires re-calculating the ruleset.

[1] These rule subsets both include the two membership rules (Section 2.2.3), making them cumulatively larger.

3.3. Removal of inefficient rules

Rule *#eq-ref* (**Code 27**) which infers that each resource is equivalent to itself, greatly bloats the dataset:

$$T(?s, ?p, ?o) \rightarrow T(?s, sameAs, ?s),$$
$$T(?p, sameAs, ?p), T(?o, sameAs, ?o)$$

Code 27. Rule inferring that each unique resource is equivalent to itself (#eq-ref).

For each unique resource, this rule creates a new statement indicating the resource's equivalence to itself. Consequently, 3 new triples are generated for each triple with unique resources, resulting in a worst-case 4x increase in dataset size (!). One could argue that there is limited practical use in materializing these statements; and it is unlikely that their absence will affect other inferences (there is one specific case where this may happen; see [30]). If needed, the rule engine could also be adapted to support them virtually. As a result, developers should at least be allowed to weigh the utility of this rule versus its computational cost. We note that some production-strength OWL reasoners, such as SwiftOWLIM, have configuration options available to disable such rules as well [27].

After applying all selections cumulatively (aside from purpose- and reference-based subsets), this leaves a ruleset of 51 rules; 18 rules less than the original ruleset. Our evaluation (Section 6) studies the performance of separately and cumulatively applying these selections.

3.4. Conformance testing

To check the conformance of our original OWL2 RL ruleset and its subset selections (Sections 3.1 – 3.3), standard OWL2 RL conformance tests should be applied. However, many test cases listed on the W3C OWL2 Web Ontology Language Conformance page for the OWL2 RL profile [52] are not actually covered by OWL2 RL (as confirmed by one of its major contributors on the W3C mailing list [77]). Therefore, we used the OWL2 RL conformance test suite presented by Schneider et al. [48]. We note that some of these tests had to be left out, either due to the limitations of the original OWL2 RL ruleset (Section 2.2; e.g., lack of datatype support), or due to difficulties testing conformance. We detail these cases in our online documentation [56].

The original OWL2 RL ruleset (Section 2.2), as well as its equivalent, conformant subsets (Sections 3.1.1 – 3.1.3), pass this conformance test suite. Regarding purpose- and reference-based subsets (Section 3.2), the final result of sequentially applying the *inference-schema* rules, *inference-instance* rules and *consistency-checking* rules also passes the conformance tests. As expected, the selections presented in Section 3.1.4 (*Equivalence with instance-based rules*) and Section 3.3 (*Removal of inefficient rules*) break conformance with the test suite; but we note that the former only loses *owl2rl-schema-completeness*.

Finally, we note that conformance of the domain-based ruleset selection (Section 3.2) cannot be checked using this test suite, since this subset only includes rules specific to the domain ontology (while the test suite checks all OWL2 RL rules). Instead, conformance of this rule subset was tested by collecting the inferences of the full ruleset (when applied on our evaluation ontologies; Section 6), and comparing them to the output of the domain-based rule subset.

4. Mobile Benchmark Framework

Fig. 1 shows the architecture overview of the MobiBench framework. For portability across platforms, this framework was implemented in JavaScript (JS), and deployed using Apache Cordova [67] (as well as JDK8 Nashorn [73]), which allows native, platform-specific parts to be plugged in. From a developer point of view, this also allows MobiBench to easily benchmark JavaScript reasoners, usable in mobile websites or cross-platform, JavaScript-based mobile apps (e.g., using Apache Cordova, Appcelerator Titanium [69]) with a write-once, deploy-everywhere philosophy, i.e., without requiring porting or transcompiling.

The core of the framework, the **Benchmark Engine**, runs benchmarks to study and compare reasoning performance on mobile platforms. The **API** supplies third parties with direct access to the MobiBench functionality, whereas the **Automation Support** is built on top of the *API*, and allows automating large numbers of benchmarks. This solution comprises two components: 1) an external *Automation Client*, which generates a (large) set of benchmark configurations based on an automation configuration, and sends them over HTTP to 2) the *Automation Web Service*, deployed on the mobile device, which invokes the *API* for each configuration and sends back the results. Since this avoids re-deploying the (rather large) MobiBench system each time on the mobile device (i.e., with new hard-coded configuration options), this setup greatly facilitates automated benchmarking. Further, it allows benchmarks to be run even when the developer has no physical access to the device. The **Analysis**

Tools aggregate benchmark results, including reasoning times and memory dumps, into CSV files.

A *RESTful Web Service*, deployed on a server (e.g., the developer's PC), comprises the **Semantic Web layer**, which supplies a uniform, standards-based interface across reasoning engines, by applying custom converters to translate standards-based rules and data; the **Selection Service**, which selects specific subsets of the OWL2 RL rule- and axiom set (Section 3.2), including domain-specific selection based on an input ontology; and the **Pre-processing Service**, to pre-process a ruleset and ontology, possibly based on the particular ontology (e.g., to support certain n-ary rules; Section 2.2.3). The Web service also includes some utility services to persist benchmark output (**Persistence Support**); by logging messages persistently (*Logging*) and storing arbitrary files (e.g., performance results, reasoning output; *File Storage*) on the server (e.g., the developer's PC).

To run a single benchmark, the *API* passes a configuration object to the *Benchmark Engine*, which (among others) specifies the reasoning engine, reasoning task, process flow and resources (i.e., rule- and datasets) to be benchmarked. A **Local Proxy** component acts as an intermediary between the mobile system and the external Web service.

The *Benchmark Engine* can perform different **reasoning tasks** (Section 4.4.2), using different **process flows** (Section 4.4.3), to better align benchmarks with existing, real-world applications. To implement the uniform engine interface, each mobile engine requires a plugin, translating the method invocations to the underlying system. Plugins for native mobile engines include both a JavaScript and native part; where bi-directional communication occurs with the native code over the Cordova bridge.

Although MobiBench is a cross-platform framework, we currently rely on Android as the mobile deployment platform, since most reasoners are either developed for Android or written in Java (which facilitates porting to Android). However, using Apache Cordova, the framework could be easily deployed on

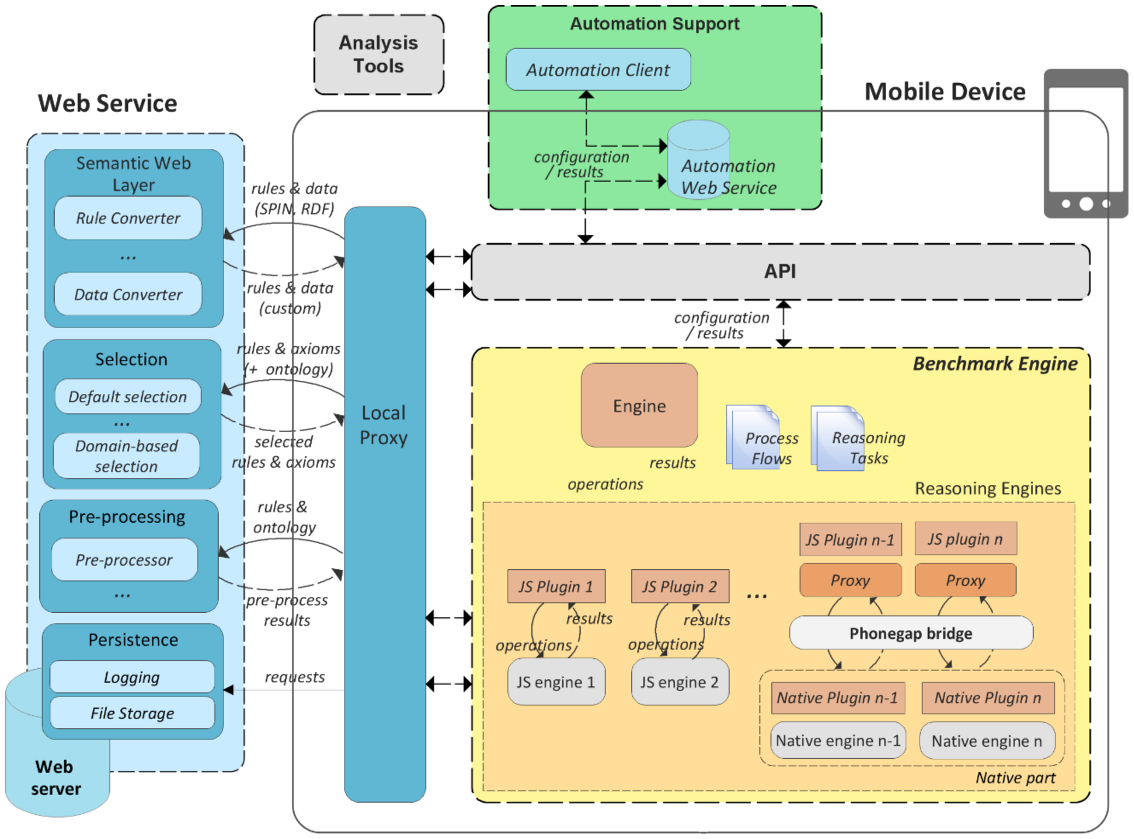

Fig. 1. MobiBench Framework Architecture.

other mobile platforms as well. The MobiBench framework can be found online [56].

In the subsections below, we elaborate on the MobiBench main components, namely the Semantic Web layer (Section 4.1), Selection Service (Section 4.2), Pre-processing Service (Section 4.3) and Benchmark Engine (Section 4.4); and indicate extension points for each component. Section 5 shows how developers can utilize the benchmark framework.

4.1. Uniform Semantic Web Layer

A range of semantic rule standards are currently being used, including the Semantic Web Rule Language (SWRL) [26], Web Rule Language (WRL) [3], Rule Markup/Modeling Language (RuleML) [12], and the SPARQL Inferencing Notation (SPIN) [35]. Some reasoning engines also introduce their own custom rule formats (e.g., Apache Jena, RDFQuery) or rely on non-Semantic Web syntaxes (e.g., Datalog: IRIS, PocketKRHyper). When benchmarking multiple systems, this multitude of formats prevents direct re-use of a single rule- and dataset. To cope with this, MobiBench supplies a uniform, standards-based resource interface, which supports SPIN and RDF as input rules and data, and dynamically translates the input to formats supported by the engines. This way, developers can re-use a single SPIN ruleset and RDF dataset across different engines.

Since the only available SPIN API is developed for the Java Development Kit (JDK) [34], the required conversion functions are deployed on an external Web service. To convert incoming SPIN rules, the SPIN API is utilized to generate an Abstract Syntax Tree (AST), which is then visited by a *Rule Converter* to convert the rule. To convert incoming RDF data, a *Data Converter* can utilize Apache Jena [4] to query and manipulate the data. The *Local Proxy* supplies local functions for remote resource conversion.

Below, we shortly discuss SPIN and our rationale for its choice as the input rule format (Section 4.4.1). Afterwards, we discuss our current converters, and how new converters can be developed and plugged into the Web service (Section 4.1.2).

4.1.1. SPIN

SPIN is a SPARQL-based rule- and constraint language, which provides a natural, object-oriented way of dealing with constraints and rules associated with RDF(S)/OWL classes. In the object-oriented design paradigm, classes define the structure of objects (i.e., attributes) together with their behavior, which includes creating/changing objects (rules) and ensuring a consistent object state (constraints). Similarly, SPIN allows directly associating locally-scoped rules and constraints to their related RDF(S)/OWL classes, using properties such as *spin:rule* and *spin:constraint*.

To serialize rules and constraints, SPIN relies on SPARQL [21], a W3C standard with sufficient expressivity to represent both queries and general-purpose rules and constraints. SPARQL is supported by most Semantic Web systems, and is well known by Semantic Web developers. As such, this rule format is more likely to be easily comprehensible to developers. Further, relying on SPIN also simplifies support for our current rule engines (see below).

4.1.2. Rule and Data Conversion

Regarding rule-based reasoners, our choice for SPIN greatly reduces conversion effort for systems with built-in SPARQL support. **RDFStore-JS** supports INSERT queries from SPARQL 1.1/Update [21], which are easy to obtain from SPIN rules in their SPARQL query syntax. Both **AndroJena** and **RDFQuery** support a triple-pattern like syntax, which likewise makes conversion from SPIN straightforward. Other rule engines lack built-in Semantic Web support, and require more significant conversion effort. Two systems, namely **PocketKrHyper** and **IRIS**, accept Datalog rules and facts in a Prolog-style input syntax. For these cases, we utilize the same first-order representation as in the W3C OWL2 RL specification [13], namely *T(?s,?p,?o)* (since predicates may also be variables, a representation such as *predicate(subject, object)* is not an option in non-HiLog).

Currently, our converters support SPIN functions that represent primitive comparators (greater, equal, etc.) and logical connectors in FILTER clauses. Advanced SPARQL query constructs, such as (not-)exists, optional, minus and union, are not yet supported, since converting them to all rule engine formats is challenging. Also, we have not required these constructs until now (i.e., not for the OWL2 RL ruleset, nor the clinical ruleset used in [59]). None of the OWL reasoners (Section 4.4.1) required (data) conversion, since they can consume serialization of OWL in RDF out of the box.

Extensibility To plug in a new resource format, developers create a new converter class implementing the uniform converter interface. The class is then added to a configuration file (*spin2s.txt* / *rdf2s.txt*), used by the Web service to dynamically load converter class definitions at startup. Each converter identifies its own format via a unique ID, allowing to match incoming conversion requests to the correct converter.

4.2. Selection Service

Due to its rule-based axiomatization, the OWL2 RL profile greatly facilitates applying subsets of axioms. In Section 3, we discuss multiple selection criteria, such as logical equivalence with other rules, and subsets based on purpose and reference. The *Selection Service* automatically performs these kinds of selections on the OWL2 RL rule- and axiom set, given one or more selection criteria. As before, since the only available API for SPIN (i.e., the input rule format) is developed for Java [34], this component is deployed on the Web service.

The *Default Selection* function selects an OWL2 RL subset, given a list of selection criteria indicating rules and axioms to leave out, replace or add. The *Domain-based Selection* function selects a minimal OWL2 RL rule subset, leaving out rules that are not relevant to a given ontology (i.e., not yielding any inferences). In doing so, we may greatly reduce the ruleset without losing expressivity.

We note that determining this domain-based ruleset manually is cumbersome and error-prone. One cannot just check whether referenced constructs are present; e.g., the ontology may contain *owl:subClassOf* constructs, but the premise of *#scm-eqc2* requires two classes to be subclasses of each other, which is less likely. Furthermore, some rules may be indirectly triggered by other rules, which means that only checking inferences per rule is insufficient as well. Tai et al. [53] describe a "selective rule loading" algorithm to determine this ruleset. As a type of "naïve" forward-chaining algorithm, it executes each rule sequentially on the initial dataset, adding resulting inferences. In case a rule yields results, it is added to the "selective ruleset" R_o^-. This process continues until no more inferences are generated. We implemented this algorithm in the *Domain-based Selection* function.

Clearly, this process should be executed each time the ontology schema is updated, but also when certain instances, constituting new data patterns, are added (e.g., reciprocal *owl:subClassOf* relations would make the *#scm-eqc2* rule relevant). Therefore, its suitability depends on the frequency and significance of such updates; i.e., whether the ontology structure is relatively "stable" or "volatile". By deploying this service directly on the mobile device, and even integrating it with the reasoner, these drawbacks could be mitigated (see future work).

Extensibility Supporting a new selection criterium depends on its requirements. In case an a priori analysis of the ontology is required, developers need to create a new subclass of the *DomainBasedSelection* class. Else, the developer simply adds a new subfolder under the *owl2rl/* folder in MobiBench, which keeps a list of rules and axioms to be removed, replaced or added.

4.3. Pre-processing Service

To support certain n-ary OWL2 RL rules, some solutions require pre-processing the ruleset and target ontology. The *Pre-processing Service* supports 3 solutions for n-ary rules from **(L2)** (see Section 2.2.3): (1) *instantiate the rules*, based on schema assertions found in the ontology; (2) *normalize (or "binarize") the input ontology*, to only contain binary versions of the n-ary assertions, and apply binary versions of the rules; and (3) *replace each rule by 3 auxiliary rules*.

When applying solutions (1) and (2), pre-processing needs to occur initially and each time the ontology is updated. Solution (3) has its own drawbacks; for each "complete" inference for an n-ary assertion (size *n*), it infers *n+1* intermediary inferences that do not follow from OWL2 RL semantics (see Section 2.2.3). The suitability of these solutions clearly depends on the scenario. As with the *Selection Service*, deploying this service directly on the mobile device would alleviate some of these drawbacks (see future work). Currently, it is deployed on the Web service, since only a Java SPIN API is available.

Extensibility To support a new pre-processing mechanism, the developer creates a new subclass of the *PreProcessor* class. In case the mechanism requires ontology analysis (cfr. solutions (1), (2)), *OntologyBasedPreProcessor* should be subclassed, which features utility methods for that purpose.

4.4. Benchmark Engine

The Benchmark Engine performs benchmarks of reasoning engines, following a particular reasoning setup. In doing so, it allows studying and comparing reasoning performance on mobile platforms. A reasoning setup includes a reasoning task and process flow. By supporting different setups, and allowing new ones to be plugged in, benchmarks can be better aligned to real-world scenarios.

In Section 4.4.1, we elaborate on the currently supported reasoning engines. Next, we discuss the available reasoning tasks (Section 4.4.2) and process flows

(Section 4.4.3), as well as the supported measurement criteria (Section 4.4.4).

4.4.1. Reasoning Engines

Below, we categorize the supported engines according to their reasoning support. The engines not indicated as Android systems, excluding the JavaScript (JS) engines, were manually ported to Android. In this categorization, we consider rule-systems as any system that can calculate the deductive closure of a ruleset, i.e., execute a ruleset and output resulting inferences (not necessarily limited to this feature).

Rule-based systems

AndroJena [66] is an Android-ported version of Apache Jena [4]. It supplies a rule-based reasoner, which supports both forward and backward chaining, respectively based on the RETE algorithm [18] and SLG resolution [14].

RDFQuery [75] is a JavaScript RDF store that performs queries using a RETE network, and implements a naïve reasoning algorithm.

RDFStore-JS [76] is a JavaScript RDF store, supporting SPARQL 1.0 and parts of SPARQL 1.1. We extended this system with naïve reasoning, accepting rules as SPARQL 1.1 INSERT queries.

IRIS (Integrated Rule Inference System) [10] is a Java Datalog engine meant for Semantic Web applications. The system relies on bottom-up evaluation combined with Magic Sets [8].

PocketKrHyper [50] is a J2ME first-order theorem prover based on a hyper tableaux calculus, and is meant to support mobile semantic apps. It supplies a DL interface that accepts DL expressions and transforms them into first-order logic.

OWL reasoners

AndroJena supplies an OWL reasoner, which implements OWL Lite (incompletely) and supports *full*, *mini* and *micro* modes that indicate custom expressivities; and an RDFS reasoner, similarly with *full*, *default* and *simple* modes. For details, we refer to the Jena documentation [68].

The **ELK** reasoner [29] supports the OWL2 EL profile, and performs (incremental) ontology classification. Further, Kazakov et al. [28] has demonstrated that it can take advantage of multi-core CPUs of modern mobile devices.

HermiT [19] is an OWL2 DL reasoner based on a novel hypertableaux calculus, and is highly optimized for performing ontology classification.

JFact [71] is a Java port of the FaCT++ reasoner, which implements a tableau algorithm and supports OWL2 DL expressivity.

Pellet [51] is a DL reasoner with sound and complete support for OWL2 DL, featuring a tableaux reasoner. It also supports incremental classification.

Our evaluation (Section 6) focuses on 2 rule-based systems, namely a native Android system (AndroJena) and a JS system (RDFStore-JS). As mentioned, from a development perspective, JS engines are especially interesting since they can be directly used by cross-platform, JS-based mobile apps (e.g., using Apache Cordova [67]). Further, we benchmarked 3 OWL2 DL reasoners, namely HermiT, JFact and Pellet. We note that an exhaustive comparison of all systems warrants its own paper, and is considered out of scope.

Extensibility To support a new JS reasoner, the developer writes a JS *plugin* object, which implements a uniform reasoner interface and specifies the accepted rule- and data format, the process subflow (if any) dictated by the engine (Section 4.4.3), and its available settings; e.g., reasoning scope (OWL, RDFS). To rule out communication, console output, etc. influencing measurements, each plugin is responsible for capturing fine-grained result times using our ExperimentTimer API. Any required JavaScript libraries, as indicated by the plugin, are automatically loaded. Developers register their plugins in an *engine.json* file.

For native engines, the developer similarly implements a native plugin class, and supplies a skeleton JS plugin. The system wraps this skeleton plugin with a proxy object, which delegates invocations to the native plugin over the Cordova bridge (see Fig. 1). In practice, native (Android) reasoners often have large amounts of dependencies, some of which may be conflicting (e.g., different versions of the same library). To circumvent this issue, we package each engine and its dependencies as jar-packaged .dex files, which are automatically loaded at runtime. For more details, we refer to our online documentation [56].

4.4.2. Reasoning Tasks

Currently, we support three reasoning tasks. Below, we shortly summarize each task. Fig. 2 illustrates the dependencies between these tasks.

1) Rule-based materializing inference: this involves computing the deductive closure of a ruleset for a dataset, and adding all inferences to the dataset.

2) OWL2 materializing inference: given an ontology, this involves materializing all inferences based on an OWL2 expressivity (e.g., OWL2 Full, OWL2 DL, OWL Lite, or some other reduced expressivity).

This task can also be performed by rule engines, using the rules axiomatizing the OWL2 RL semantics. Fig. 2 shows two types of OWL inference: *"built-in"* inference of any kind (e.g., OWL2 DL, QL, Lite, etc.), which only requires an input ontology; and *OWL2 RL reasoning*, which uses a rule engine and accepts both an OWL2 RL ruleset and ontology as input.

Regarding our choice for materializing inferences vs. reasoning per query (e.g., via resolution methods such as SLG [14]), we note that each have their advantages and drawbacks on mobile platforms. Prior to data access, the former involves an expensive pre-processing step that may significantly increase the dataset scale, which is problematic on mobile platforms, but then leaves query answering purely depending on speed of data access. In contrast, the latter incurs a reasoning overhead for each query that depends on dataset scale and complexity. Another materialization drawback is that inferences need to be (re-)computed whenever new data becomes available. For instance, Motik et al. [41] combine materialization with a novel incremental reasoning algorithm, to efficiently update previously drawn conclusions. To allow benchmarking such incremental methods, our framework supports an "incremental reasoning" process flow (Section 4.4.3). For the purposes of this paper, we chose to focus on a materialization approach, although supporting resolution-based reasoning is considered future work. We note that many Semantic Web rule-based reasoners, including DLEJena [40], SAOR [25], OwlOntDb [17] and RuQAR [5], also follow a materialization approach.

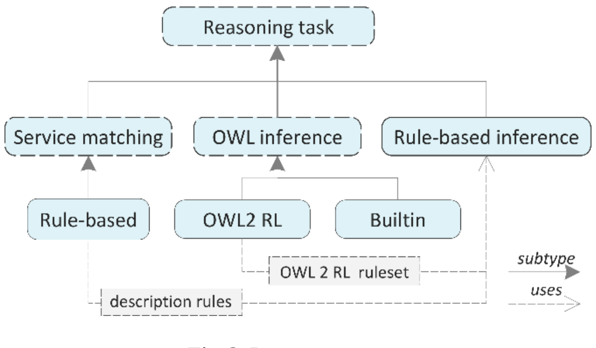

Fig. 2. Reasoning types.

3) Service matching: checks whether a user goal, which describes the services the user is looking for, matches a service description. In its rule-based implementation, a pre- or post-condition / effect from one description (e.g., goal) acts as a rule; and a condition from the other (e.g., service) serves as a dataset, which is done by "freezing" variables, i.e., replacing them by constants. A match is found when rule execution infers the consequent. This rule-based task can be enhanced with ontology reasoning, by including an OWL2 RL ruleset with the match rule(s). Our reason for focusing on service matching stems from our mobile setting; e.g., it enables mobile apps to locate useful services in a smart environment, with all necessary computation taking place on the mobile platform (see e.g., [58]).

Extensibility Reasoning tasks are implemented as JS classes, with a hierarchy as shown in Fig. 2. A new reasoning task class needs to implement the *inference* function, which realizes the task by either directly invoking the uniform reasoner interface (see Section 4.4.1), delegating to another task class (e.g., *Rule-based inference*) or to a subflow (see Section 4.4.3 – *Extensibility*). The *Reasoning task* super class provides functions such as checking conformance, collecting result times, and logging inferences. A new task file and constructor should be listed in *tasks.json*.

4.4.3. Process Flows

To better align benchmarks with real-world use cases, MobiBench supports several process flows, which dictate the times at which operations (e.g., load data, execute rules / perform reasoning) are performed. From previous work [59], [60], and in line with our choice for materializing inferences, we identified two useful process flows:

Frequent Reasoning: in this flow, the system stores all incoming facts directly in a data store (which possibly also includes an initial dataset). To generate new inferences, reasoning is periodically applied to the entire datastore. Concretely, this entails loading a reasoning engine with the entire datastore each time a certain timespan has elapsed, applying reasoning, and storing new inferences into the datastore.

Incremental Reasoning: here, the system applies reasoning for each new fact (currently, MobiBench only supports monotonic reasoning, and thus does not deal with deletions). In this case, the reasoning engine is first loaded into memory (possibly with an initial dataset). Then, reasoning is (re-)applied for each incoming fact, whereby the new fact and possible inferences are added to the dataset. Some OWL reasoners directly support incremental reasoning, such as ELK and Pellet. As mentioned, Motik et al. [41] implemented an algorithm to optimize this kind of reasoning, initially presented by Gupta et al. [23].

Further, we note that each reasoner dictates a *sub-flow*, which imposes a further ordering on reasoning operations. In case of OWL inference (implemented via e.g., tableau reasoning), data is typically first loaded into the engine, and then an inference task is performed (*LoadDataPerformInference*). Similarly, *RDFQuery*, *RDFStore-JS* and *AndroJena* first load data and then execute rules. For the *IRIS* and *Pocket-KrHyper* engines, rules are first loaded (e.g., to build the Datalog KB), after which the dataset is loaded and reasoning is performed (*LoadRulesDataExecute*). For more details, we refer to previous work [59].

Extensibility Process flows are implemented as JS classes. Each main process flow is listed in *flows.json*, and will call a reason task at certain times (e.g., frequent vs. incremental) and with particular parameters (e.g., entire dataset vs. new fact). A subflow is specific to a particular reasoning task (see Section 4.4.2 – *Extensibility*). A *Reason task* may thus utilize a subflow class behind-the-scenes, in case multiple subflows are possible. When called, a subflow class executes the uniform reasoning functions (e.g., load-data, execute) in the appropriate order.

4.4.4. Measurement Criteria

The Benchmark Engine allows studying and comparing the metrics listed below.

Performance:

Loading times: time needed to load data and rules, ontologies, etc. into the engine.

Reasoning times: time needed to infer new facts or check for entailment.

Memory consumption: total memory consumed by the engine after reasoning. Currently, it is not feasible to measure this criterium for non-native engines; we revisit this issue in Section 6.3.

Conformance:

The *Benchmark Engine* allows to automatically compare inferences to the expected output for conformance checking (Section 3.4). As such, MobiBench allows investigating the completeness and soundness of inference as well (cfr. [22]).

Other related works focus on measuring the fine-grained performance of specific components, such as large joins, Datalog recursion and default negation [38]. In contrast, the goal of MobiBench is to find the most suitable reasoner on a mobile platform, given a particular application scenario (e.g., reasoning setup, dataset). Our performance metrics support this objective. Finally, we note that the performance of the Semantic Web layer (Section 4.1), domain-based selection (Section 4.2) and pre-processing (Section 4.3) are not measured. The Semantic Web layer will not be included in actual reasoning deployments, and only aims to facilitate benchmarking. Because of the current drawbacks of domain-specific selection and pre-processing (e.g., difficulty with volatile ontologies), and the current inability to deploy these services directly on the mobile platform, we do not measure their performance. Improving and optimizing these services, by e.g., directly integrating them with the reasoner, is considered future work.

5. Using MobiBench for Benchmarking

While the previous section indicated how MobiBench can be extended by third-party developers, this section describes how developers can utilize MobiBench for benchmarking. Developers may run benchmarks programmatically (Section 5.1) or use the automation support (Section 5.2). To aggregate benchmark results into summary CSV files, developers can utilize the analysis tools (Section 5.3). For more detailed instructions, we refer to our online documentation [56].

5.1. Programmatic Access

To execute benchmarks programmatically, developers call the MobiBench's *execBenchmark* function with a configuration object, specifying options for reasoning and resources. Below, we show an example (Code 28):

```
config: {
  engine: 'androjena', nrRuns: 10, warmupRun: true,
       dumpHeap: true,
  reasoning: {
    task: ‘ontology_inference',
    mechanism: {
      ontology_inference: {
        type: 'owl2rl', dependency: 'rule_inference'
      },
      rule_inference: {
        mainFlow: 'frequent',
        subFlow: 'load_data_exec_rules'
      } } },
  resources: {
    ontology: {
      path: 'res/owl/data/0.nt',
      type:'data', format:'RDF', syntax:'N-TRIPLE'
    },
    owl2rl : {
      axioms: {
        path: 'res/owl/owl2rl/full/axioms.nt',
        type:'data', format:'RDF', syntax:'N-TRIPLE'
      },
      rules: {
        path: 'res/owl/owl2rl/full/rules.spin',
        type: 'rules', format: 'SPIN' },
     preprocess: 'inst-rules',
    selections: [ 'inf-inst', 'entailed' ]
  },
  confPath: 'res/owl/conf/ontology_inference/0.nt'
  outputInf: 'res/output/ontology_inference/...'
  id: '...' }
```

Code 28. Example benchmark configuration object.

This object specifies the unique engine id, the number of experiment runs, possibly including a "warmup" run (not included in the collected metrics), and whether memory usage should be measured (*dumpHeap*). The *reasoning* part indicates the high-level reasoning task (i.e., *ontology_inference*) and concrete mechanism (i.e., *owl2rl*), as well as details on dependency tasks (i.e., *rule_inference*), including its main and sub process flow.

The *resources* section lists the resources to be used in the benchmark; in this case, an ontology and OWL2 RL axioms and rules. Further, the section specifies that the *inst-rules* pre-processing method (i.e., instantiate rules; Section 2.2.3, (1)) should be applied, as well as selections *inf-inst* (i.e., *inference-instance* subset) and *entailed* (i.e., leaving out logically redundant rules) (Section 3). Both involve calling the respective services on the Web service. The section may also indicate the path for storing inferences (*outputInf*); as well as the expected reasoning output (*confPath*), to allow for automatic conformance checking.

5.2. Automation Support

Due to the potential combinatorial explosion of configuration options, including engines and their possible settings, resources and OWL2RL subsets, manually writing configurations quickly becomes impractical. For that purpose, we implemented an *Automation Support* component.

This solution includes an *Automation Client*, deployed on a server or PC, which generates a set of benchmarks based on an automation configuration; and communicates over HTTP with the *Automation Web Service* on the mobile device, which locally invokes the MobiBench *API* and returns the benchmark results. In the *Automation Client* code, developers specify ranges of configuration options, whereby each possible combination will be used to run a benchmark. Code 29 shows (abbreviated) example code for running a set of OWL2 RL benchmarks:

```
OWL2RLRunConfig config = new OWL2RLRunConfig();
config.setTask("owl_inference", "owl2rl");
config.select({ "entailed" },
    { "inf-inst", "entailed", "domain-based" });
config.addDataset("ore", 0, 188); ...
```

Code 29. Example automation configuration.

In this case, one subset leaves out entailed, logically redundant rules (*entailed*), and the second applies the *inf-inst* (i.e., inference-instance subset), *entailed* and *domain-based* (i.e., selecting a domain-based subset) selections. Both rulesets are applied on all benchmark ontologies, creating a total of 378 benchmarks.

5.3. Analysis Tools

To deal with large amounts of benchmark results, the MobiBench *Analysis Tools* assemble benchmark results into a CSV file. This file lists the performance results and memory usages per configuration; including process flow and reasoning task, rule subsets, engine-specific options, and datasets.

Further, the *Analysis Tools* include a utility function to compare performance times of two reasoning configurations (e.g., different OWL2 RL subsets), and output both the individual (i.e., per benchmark ontology) and total (i.e., aggregated) differences in performance. The *Analysis Tools* are available both as source code and a command line utility. See our online documentation [56] for more info.

6. Mobile Reasoning Benchmark Results

This section presents benchmark results for materializing ontology inferences and executing semantically enhanced, rule-based service matching on mobile platforms, obtained using MobiBench.

6.1. Reasoning Tasks

Our benchmarks cover the tasks listed below. We note that, although rule-based reasoning is not benchmarked separately, it is used to implement OWL2 RL reasoning and (rule-based) service matching.

6.1.1. OWL2 Materializing Inference

Regarding the OWL2 reasoning task, which involves materializing ontology inferences (Section 4.4.2), benchmarking goals include 1) measuring the performance impact of different OWL2 RL subset selections (Section 3); 2) benchmarking two rule-based systems (AndroJena, RDFStore-JS) with best-performing OWL2 RL rulesets, as well as three OWL2 DL reasoners (HermiT, Pellet, JFact). To find the best-performing OWL2 RL ruleset, we consider the following orthogonal cases: "stable" vs. "volatile" ontologies (i.e., whether they are subject to frequent and significant changes; Section 4.2); and OWL2 RL-conformant vs. non-conformant rulesets.

Currently, we chose to only apply the *Frequent Reasoning* process flow; since most systems either support incremental reasoning only partially (e.g., Pellet: only incremental classification), or not at all. This means they will have virtually identical performance for incremental reasoning steps.

6.1.2. Semantically-Enhanced Service Matching

Our goal includes studying the utility and feasibility of leveraging OWL2 RL for semantic reasoning on mobile platforms. Aside from the stand-alone materialization of ontology inferences (Section 6.1.1), another use case involves semantically enhancing rule-based tasks. In particular, rule-based service matching involves executing a pre- or post-condition (e.g., from a goal) as a rule on another condition (e.g., from a service), and vice-versa. By extending the service matching rules with an OWL2 RL ruleset, we can enhance this task with ontology-based reasoning while using only a single component (Section 4.4.2). Below, we give an example of semantically-enhanced matching.

One of the user goals features a pre-condition that, given a person (type *Person*), book (type *Book*) and credit card account (type *CreditCardAccount*), a service should return a price (type *Price*). A candidate service mentions a pre-condition that, given the same input, a tax-free price (type *TaxFreePrice*) is returned. Semantically-enhanced service matching correctly infers that the service output also has type *Price*, since *TaxFreePrice* is a subclass of *Price;* thus producing a match. In conceptual terms, the user goal requests a *Price*, which also includes *TaxFreePrice*'s. In the inverse direction, service output *TaxFreePrice* does not comprise goal output *Price*, and thus does not match.

This benchmark aims to measure and compare the computational cost of the original task with its semantically enhanced version. For this purpose, we reuse the best-performing rule engine and OWL2 RL subsets, as determined by Section 6.1.1.

6.2. Benchmark Resources

To benchmark our reasoning tasks, we rely on the validated resources listed below (available for download at our online documentation [56]).

6.2.1. OWL2 Materializing Inference

This section lists resources for OWL2 inference, including ontologies (Section 6.2.1.1) and rulesets for OWL2 RL reasoning (Section 6.2.1.2).

6.2.1.1 OWL2 Ontologies

OWL 2 RL Benchmark Corpus [39]: Matentzoglu et al. extracted this corpus from general-purpose repositories including the Oxford Ontology repository [74], the Manchester OWL Corpus (MOWLCorp) [72], and BioPortal [54], a comprehensive repository of biomedical ontologies. The corpus contains ontologies from clinical and biomedical fields (ProPreo, ACGT, SNOMED), linguistic and cognitive engineering (DOLCE) and food & wine domains (Wine), thus covering a range of use cases for general-purpose, ontology-based reasoning.

To suit the constrained resources of mobile platforms, we extracted ontologies with 500 statements or less from this corpus, resulting in 189 benchmark ontologies (total size: ca. 9Mb). By focusing on OWL2 RL ontologies, all ontology constructs are supported by all evaluated reasoners, i.e., OWL2 RL and DL.

In Section 6.5.1.1, the benchmark ontologies are ordered 0–188, with an ontology's cardinal number indicating its relative OWL2 RL reasoning performance.

6.2.1.2 OWL2 RL Rulesets

To study the effects of OWL2 RL subset selections on performance (Section 3), we created multiple benchmark rulesets using our *Selection Service* (Section 4.2). We summarize each selection below, and list their label used in the benchmark results. Note that, when discussing the benchmark results, the "+" symbol indicates applying one or more selections on the OWL2 RL ruleset.

Selections from (1) still guarantee OWL2 RL conformance, as summarized in Section 3.4. Moreover, the *inst-ent* selection from (2) still guarantees *owl2rl-instance-completeness* (Section 3.1.4).

(1) Conformant selections

- *entailed*: leave out logically redundant rules (Section 3.1.1);
- *extra-axioms*: add extra supporting axioms, which allows leaving out specific rules (Section 3.1.2);
- *gener-rules*: add generalized rules, each replacing two or more specialized rules (Section 3.1.3);
- *inf-inst*: retain inference rules referring to both instance and schema elements (Section 3.2);
- *inf-schema*: retain inference rules referring only to schema elements (Section 3.2);
- *consist*: retain only consistency-checking rules (Section 3.2);

- *domain-based*: leave out rules not referenced by the ontology (Section 3.2).

(2) Non-conformant selections

- *inst-ent*: leave out schema-based rules not yielding extra instance inferences (Section 3.1.4);
- *ineff:* leave out inefficient rules (Section 3.3).

To support n-ary rules from (L2) (Section 2.2.3) we chose to only apply solution (1), i.e., *instantiating the ruleset*. This was done for all benchmarks, i.e., all benchmark results were obtained with a ruleset that can deal with all n-ary rules. Since the benchmark ontology corpus (Section 6.2.1.1) only contains 18 intersections in total (with no property-chain or has-key assertions), we chose a solution that, due to its particular process, leaves out these rules in case no related n-ary assertions are found. Due to the low number of relevant assertions in this corpus, comparing the performance impact of different solutions would not make much sense (this is considered future work). We also note that ontologies with intersections were extended with relevant instance assertions, so inferences would be made based on the (instantiated) #*cls-int1* rule.

6.2.2. Semantically-Enhanced Service Matching

OWL-S Service Retrieval Test Collection [32]: this collection contains 42 goals and 1083 services for OWL-S. For the purposes of our benchmark, we extracted pre- and post-conditions / effects (originally in SWRL) in the form of SPIN rules and RDF data, including the types of input and output variables. Since not all descriptions contained such conditions, this resulted in a final set of 17 goals and 152 services.

Further, we generated an extended version of this dataset that includes all related ontology elements, making each condition self-contained to facilitate semantic service matching. This was done by manually analyzing the conditions and referenced ontologies, and only including elements affecting OWL2 RL inferences. Seeing how only avg. ca. 5 ontology terms are referenced per condition, it would have been excessive to include referenced ontologies in their entirety (with avg. ca. 2100 statements, ranging between ca. 30 to ca. 40k statements).

6.3. Benchmark Measurements

Benchmarks capture the metrics discussed in Section 4.4.4, including loading and reasoning times and memory consumption. Regarding memory, Android Java heap dumps are used to accurately obtain memory usage of native Android engines. However, regarding JavaScript engines, heap dumps can only capture the entire memory size of the native WebView (used by Apache Cordova to run JavaScript on native platforms), not individual components inside it. Although Chrome DevTools [70] is more fine-grained, it only records heap allocations inside the mobile Chrome browser. Therefore, memory measurements were only possible for native Android reasoners.

6.4. Benchmark Hardware

To perform the benchmarks, we used an LG Nexus 5 (model LG-D820), with a 2.26 GHz Quad-Core Processor and 2Gb RAM. This device runs Android 6, which grants Android apps 192Mb of heap space. During the experiments, the device was connected to a power supply.

6.5. Benchmarking Results and Discussion

This section presents and discusses the benchmark results for OWL materializing inference (Section 6.5.1) and service matching (Section 6.5.2).

6.5.1. OWL Materializing Inference

First, we show the results for individually benchmarking OWL2 RL ruleset selections (Section 6.5.1.1). Based on this analysis, Section 6.5.1.2 presents the best performing OWL2 RL rule subsets, given different requirements and scenarios, and sets them side by side with benchmark results of OWL2 DL reasoners (HermiT, JFact and Pellet). Unless indicated otherwise, result times include ontology loading, reasoning, and inference collection.

6.5.1.1 OWL2 RL Ruleset Selections

Figures 3-5 show the performance of OWL2 RL ruleset selections for AndroJena. Fig. 3 shows that leaving out logically redundant rules (+*entailed*, i.e., applying the *entailed* selection) has a slight positive impact on performance (avg. ca. -180ms), whereas also replacing specific rules by extra axioms and general rules (+ *entailed, extra-axioms, gener-rules*) performs slightly worse (avg. ca. +180ms). This was a possibility, since this selection introduces more general, i.e., less constrained, rules (e.g., less able to leverage internal data indices). Applying a domain-specific rule subset (+*entailed, domain-based*) supplies a much larger performance increase (avg. ca. -0,78s). The *inf-inst* selection improves performance even more (avg. ca. -1s). The *ineff* selection loses completeness but shows the highest gain (avg. ca. -1,3s).

Although the *inf-inst* selection shows promise, it requires materializing schema inferences using the *inf-*

schema subset, initially and in case of ontology updates. Also, when consistency needs to be checked, the *consist* ruleset needs to be separately executed. Next, we discuss the performance of *inf-schema* and *consist*, as well as the effect of ruleset selections on *inf-inst*.

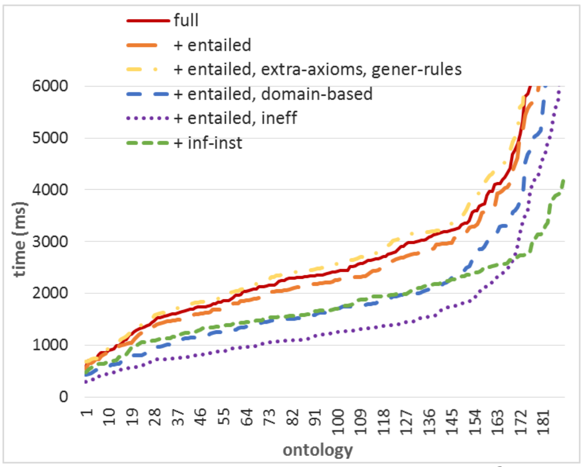

Fig. 3. AndroJena: OWL2 RL selections (*full*)[2].

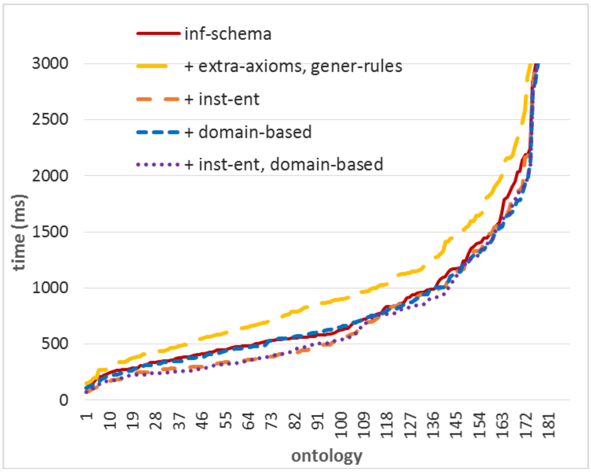

Fig. 4. AndroJena: OWL2 RL selections (*inf-schema*).

Fig. 4 shows the performance of materializing inferences in the ontology (*inf-schema*). As was the case before, ruleset selections may be applied on this subset. Similar to the *full* case, replacing specific rules with extra axioms and general rules (+*extra-axioms, gener-rules*) reduces performance (avg. ca. +250ms, compared to *inf-schema*). For *inf-schema*, a non-conformant selection is leaving out rules inferring schema inferences that do not yield extra instances (*inst-ent*, Section 3.1.4), which slightly improves performance (avg. ca. -80ms). Since *entailed* and *ineff* do not include schema-only rules, they cannot be applied here. Applying *domain-based*, alone and when combined with *inst-ent* (+*inst-ent, domain-based*), similarly improves performance slightly (avg. ca. -50ms and -100ms, respectively). However, when applying *domain-based* on the *inf-schema* subset, the *domain-based* selection needs to reconstruct the *inf-schema* ruleset for each ontology update; and the ruleset is then utilized only once[3], i.e., to materialize schema inferences in the updated ontology. Its suitability here thus depends on the performance of the *domain-based* selection, which is not measured in these benchmarks as it is deployed on a Web service. (Future work involves studying mobile deployment, see Section 8.)

After materializing the ontology with schema inferences, instance-related rules (*inf-inst*) are applied whenever new instances are added. When consistency needs to be checked, the *consist* ruleset selection is applied on a materialized set of schema and instance assertions (avg. ca. 420ms). We note that the only applicable selection here, i.e., *gener-rules*, results in very similar performance (avg. ca. 430ms).

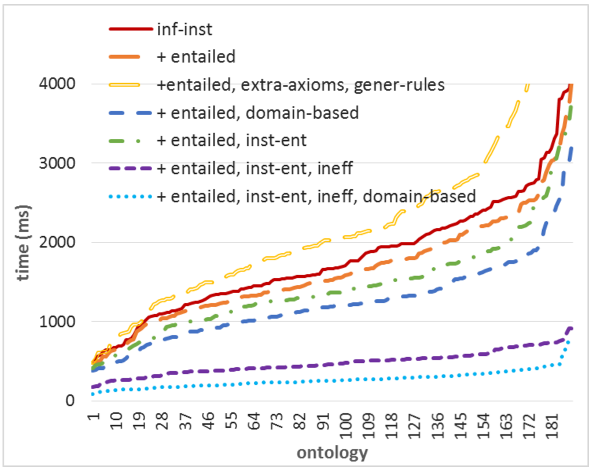

Fig. 5. AndroJena: OWL2 RL selections (inf-inst).

[2] Some figures chop off peaks to avoid skewing the graph. The full average results can be found at [56].

[3] Except for scenarios where e.g., the ontology needs to be re-materialized at each startup.

Fig. 5 shows that, similar to the *full* case, leaving out redundant rules (+*entailed*) results in small improvements (avg. ca. -145ms, compared to *inf-inst*). Additionally replacing specific rules by extra axioms and general rules (+ *entailed*, *extra-axioms*, *gener-rules*) similarly leads to performance loss (avg. ca. +0,5s), while selecting a domain-based subset (+*entailed*, *domain-based*) results in gains (avg. ca. -0,5s). Regarding non-conformant cases, a first option is to execute the rule subset on the ontology materialized by *inst-ent*, which is smaller since it lacks certain schema elements (i.e., not yielding extra instances). This scenario (+ *entailed*, *inst-ent*) improves performance by avg. ca. -340ms. Additionally removing inefficient rules (+ *entailed*, *inst-ent*, *ineff*) increases performance by avg. ca. -1,3s. Combining all selections yields reductions of avg. ca. -1,5s.

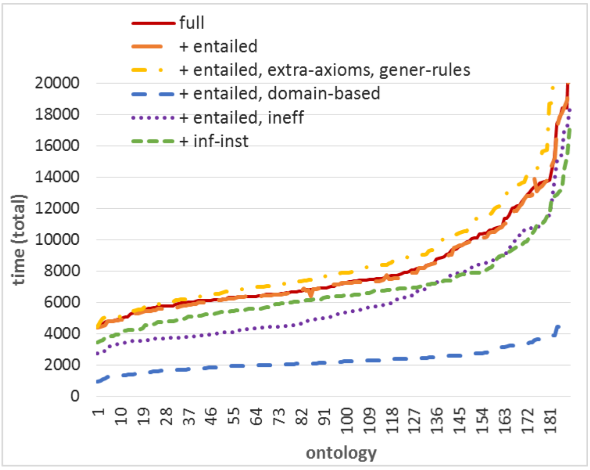

Fig. 6. RDFStore-JS: OWL2 RL selections (*full*).

Figures 6-8 show OWL2 RL subset performances for RDFStore-JS. Fig. 6 shows that, similar to AndroJena, *entailed* yields only slightly better performance (avg. ca. -100ms), whereas *entailed*, *extra-axioms* and *gener-rules* collectively result in worse performance (avg. ca. +0,85s). At the same time, compared to AndroJena, also applying *domain-based* yields much higher performance gains (avg. ca. -5,8s), while *inf-inst* (avg. ca. -1,3s) and *ineff* (avg. ca. -1,9s) have a smaller comparative impact.

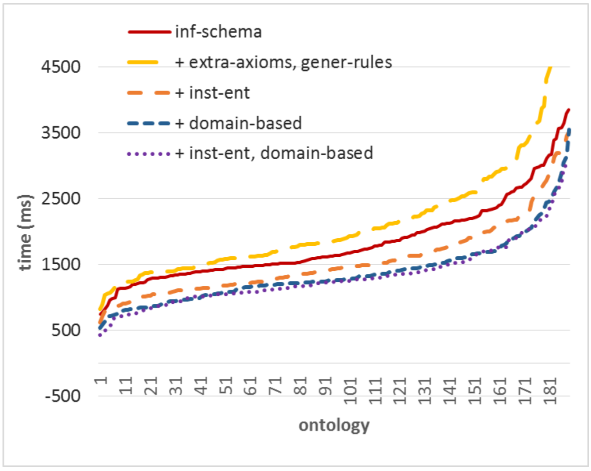

Fig. 7. *RDFStore-JS*: OWL2 RL selections (*inf-schema*).

Fig. 7 shows the performance of materializing inferences in the domain ontology (*inf-schema*). As for AndroJena, replacing specific rules (+*extra-axioms*, *gener-rules*) reduces performance (avg. ca. +380ms, compared to *inf-schema*), while leaving out "instance-redundant" rules (*inst-ent*) improves performance to a larger extent (avg. ca. -270ms). As before, we note that *entailed* and *ineff* are not applicable here. Utilizing *domain-based*, individually and combined with *inst-ent* (+*inst-ent*, *domain-based*) results in the largest improvements in performance (avg. ca. -0,46s and -0,5s, respectively), although, as mentioned, the suitability of *domain-based* could be questioned here.

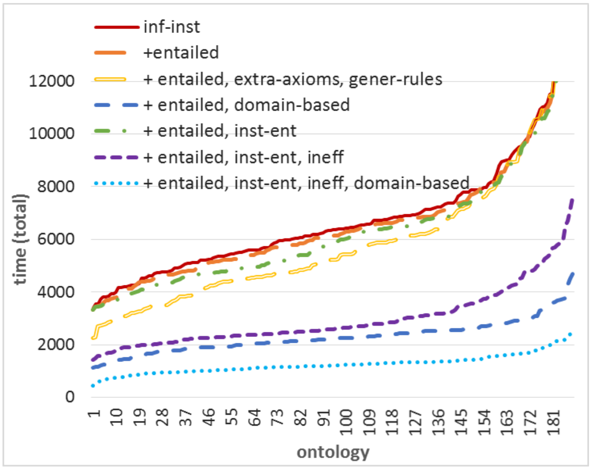

Fig. 8. *RDFStore-JS*: OWL2 RL selections (*inf-inst*).

Fig. 8 shows the results of the *inf-inst* rule subsets, applied on an ontology materialized with schema inferences. In contrast to AndroJena and the *full* case for RDFStore-JS, collectively applying *entailed*, *extra-axioms* and *gener-rules* improves performance (avg. ca. -0,8s), and exceeds the performance gained by only *+entailed* (avg. ca. -180ms). Similar to *full* (Figure 6), the *domain-based* selection (*+entailed, domain-based*) performs much better (avg. ca. -4,5s). Considering non-conformant selections, applying the rule subset on the ontology materialized via *inst-ent* (*+entailed, inst-ent*) increases performance by avg. ca. -430ms (compared to *inf-inst*). Also applying the *ineff* selection (*+entailed, inst-ent, ineff*) significantly improves performance (avg. ca. -3,8s). Combining all selections reduces reasoning times by avg. ca. -5,5s.

Finally, the *consist* ruleset yields a performance of avg. ca. 2,1s, with +*gener-rules* (only applicable selection) performing slightly better (avg. ca. -160ms).

Summary

Overall, the *entailed* selection has a relatively small performance impact, with reductions from -1,2% (rdfstore-js: *full*) to -8% (androjena: *inf-inst*). Utilizing *extra-axioms* and *gener-rules* typically results in (slightly) worse performance; which is not wholly unexpected, seeing how it replaces specific rules with more general ones (e.g., with more joins and less ability to leverage internal data indices). In some cases however, these selections perform better: i.e., when executing *inf-inst* (-21%) on RDFStore-JS.

In case of a stable ontology, additional OWL2 RL-conformant optimization options exist. Executing the *inf-inst* ruleset on a materialized ontology results in performance increases from -17% (rdfstore-js) to -36% (androjena) compared to the *full* ruleset. Here, applying the best-performing, conformant selection (i.e., *inf-inst+entailed+domain-based)* yields huge optimizations, up to -72% (rdfstore-js) compared to the original, non-selection case.

In case the conformance requirement is dropped, even larger optimizations are possible. Employing the *inst-ent* selection yields slight improvements in performance for *inf-schema*; 8% (androjena) and 15% (rdfstore). Re-using the smaller materialized ontology optimizes the *inf-inst* selection as well, up to -12% (androjena). Putting it all together, selection +*inf-inst, entailed, inst-ent, domain-based, ineff* yields dramatic improvements, as far as -90% (androjena) compared to the *full* case.

6.5.1.2 Best Overall Performance

Table 1 shows the best-effort performances of the rule engines: for the full, original OWL2 RL ruleset (*original*, for reference); when applying best-performing conformant (*conformant*) and non-conformant (*non-conformant*) rule subsets; and for cases where the domain ontology frequently (and significantly) changes (*volatile ontology*), ruling out certain selections, and cases where such changes are not likely to occur (*stable ontology*). In the latter case, times for a priori materializing the ontology (*inf-schema*), inferring new instances (*inf-inst*), and consistency checking times (*consist*) are shown as well. Based on benchmark results from the previous section, we chose the best-performing ruleset selections for each case (see table). Both total times and constituent loading and reasoning times are indicated. Further, the table sets these results side by side with the overall performance of HermiT, Pellet and JFact, well-known OWL2 DL reasoners. These reasoners perform reasoning with higher complexity (OWL2 DL), which yields extra schema (TBox) inferences not covered by the OWL2 RL rule axiomatization [37], [41]. We confirmed that the OWL2 RL and OWL2 DL reasoners infer the same ABox inferences. Clearly, any comparison should take this schema incompleteness issue into account.

In line with expectations, the table shows that AndroJena, as a native Android system and featuring a non-naïve, RETE-based forward chainer, greatly outperforms RDFStore-JS, which we manually outfitted with naïve reasoning (Section 4.4.1). As shown before, the ruleset selection suiting volatile ontologies and guaranteeing conformance (*entailed*) performs only slightly better. However, if the ontology is considered stable, the conformant *inf-inst* selection supplies huge relative gains (avg. ca. 1,6s (55%) – 5,8s (72%)) compared to the *original* case, respectively for AndroJena and RDFStore-JS (percentage indicates the proportion of time gained w.r.t. the original). At the same time, *inf-schema* yields a comparatively lower, but certainly not negligible, overhead, which is incurred for each ontology update. As mentioned, since applying the *domain-based* selection on *inf-schema* would not be advantageous in most scenarios, it is not applied here. In contrast, the best-performing conformant *inf-inst* ruleset requires the *domain-based* ruleset selection, which needs to be re-calculated for each ontology update and thus adds an extra overhead (not included here). As a result, this configuration is suitable for "stable" scenarios, where ontology updates are infrequent. Similarly, the cost of *consist* is not negligible; the frequency of applying the ruleset depends on the application scenario.

Table 1. Best overall performances (avg) (ms)

OWL2 RL*						OWL2 DL**	
AndroJena	**original**	2819 (88 | 2731)				**Hermit**	21111
		volatile ontology	**stable ontology**				
	conformant	*full*	*inf-schema*	*inf-inst*	*consist*		
		2639 (90 | 2549) + entailed	1001 (69 | 932)	1245 (187 | 1058) + entailed, domain-based	418 (195 | 223)		
	non-conformant	*full*	*inf-schema*	*inf-inst*		**Pellet**	6978
		1547 (93 | 1455) + entailed, ineff	919 (65 | 854) inst-ent	272 (165 | 106) + entailed, domain-based, ineff, inst-ent			
RDFStore-JS	**original**	8120 (618 | 7502)					
		volatile ontology	**stable ontology**			**JFact**	7034
	conformant	*full*	*inf-schema*	*inf-inst*	*consist*		
		8022 (620 | 7402) + entailed	1831 (536 | 1296)	2304 (566 | 1738) + entailed, domain-based	1947 (1282 | 665)		
	non-conformant	*full*	*inf-schema*	*inf-inst*			
		6168 (583 | 5586) + entailed, ineff	1561 (511 | 1050) + inst-ent	1255 (1080 | 176) + entailed, domain-based, ineff, inst-ent			

* : [*total-time*] ([*load-time*] | [*reason-time*] ; applied selections are shown, if any.
**: *total-time*

When dropping conformance, we find performance improvements even for volatile ontologies (avg. ca. 1,3s (45%) – 1,9s (24%)). For non-conformant reasoning in stable ontologies, the performance gain of *inf-inst* is tremendous (avg. ca. 2,5s (90%) – 6,9s (85%)). Regarding OWL2 DL reasoners, Pellet and JFact have comparable mobile performance (around avg. ca. 7s) with HermiT being a clear outlier (avg. ca. 21s).

Table 2 shows memory usage for each engine (aside from the JavaScript-based RDFStore-JS; see Section 6.3). JFact uses the least amount of memory, i.e., only 585Kb, making it a suitable choice overall (see Table 1) for mobile platforms. Nevertheless, all memory usages appear acceptable (at least on Android), seeing how each Android app receives a 192Mb max. heap.

Table 2: Memory usage (Kb)

AndroJena	HermiT	Pellet	JFact
6242	13543	12832	585

In conclusion, depending on the application scenario and requirements for full conformance, OWL2 RL reasoning can be greatly optimized on mobile platforms, making it a viable option for ontology-based reasoning. We note that, even for JavaScript systems outfitted with naïve reasoning, large performance improvements are possible. In case an application has need for extra OWL2 DL expressivity, JFact or Pellet may be used, albeit at significantly lower performance.

6.5.2. Semantically-Enhanced Service Matching

Table 3 presents the performance of rule-based semantic service matching by the best-performing rule engine (i.e., AndroJena), showing the average total time of a service match, which includes matching the pre- and post-condition of a user goal to a service, and vice-versa.

Table 3: service match performance using AndroJena (avg) (ms)

	original	*OWL2 RL*		
		full	*conf*	*non-conf*
total (ms)*	26 (11 | 15)	1062 (50 | 1012)	954 (48 | 906)	441 (47 | 394)
			domain-based	
			754 (49 | 705)	280 (48 | 232)
# matches**	*precond*			
	g>s: 22, **s>g**: 23	**g>s**: 32, **s>g**: 53		
	effect			
	g>s: 3, **s>g**: 5	**g>s**: 4, **s>g**: 14		

*: [*total-time*] ([*load-time*] | [*reason-time*]
**: number of matches per direction (e.g., g>s = goal > service)

The table shows the original, non-enhanced case (*original*), and when enhanced with ontology reasoning *(OWL2 RL*). In particular, when applying the full ruleset (*full*); a conformant (*conf;* i.e., *+entailed*) and non-conformant subset (*non-conf*; i.e., *+entailed, inst-ent, ineff*); and the *domain-based* selection.

A total of 50 extra matches are found by semantically enhancing this task, mostly by leveraging subclass hierarchies (for an example, see Section 6.1.2). A full list of extra matches can be found online [56]. While the performance of *original* is reasonable, the average reasoning time for *full* is almost two orders of magnitude larger. Applying a conformant, non domain-specific rule subset yields only a slight (ca. 10%) improvement. The non-conformant ruleset performs much better, improving performance by ca. 59%. As expected, the *domain-specific* selection yields larger performance increases, respectively ca. 29% and 74%. However, to calculate the domain-specific ruleset, this selection requires access to all (or at least, representative) user goals / services and their related schema, which may not be possible in practice.

From these results, we can conclude that semantically-enhanced service matching, with its potential to increase the amount of valid matches, has a distinct utility. Depending on application constraints, the total performance overhead per service match (including matching pre- and post-conditions, and in both directions) ranges from ca. 0,95s to 0,28s.

7. Related Work

In the state of the art on rule-based OWL reasoning, most works focus on separating TBox from ABox reasoning [5], [17], [25], [40], [41]. In most cases, a separate OWL reasoner is utilized to compute and materialize schema inferences [5], [17], [40]. However, this is inadvisable on mobile platforms, since it necessitates deploying two (resource-heavy) reasoner systems, i.e., an OWL reasoner and rule engine. After this separate schema reasoning step, some works [5], [40], [41] proceed with a rule-template approach; where OWL2 RL rules are instantiated based on the materialized input ontology. In particular, multiple instantiated rules are created for each rule, whereby schema variables are replaced by concrete schema references. We support a similar solution to support certain n-ary rules, and applied it in our benchmarks. Implementing and benchmarking this as an optimization for all rules is considered future work.

Tai et al. [53] propose a selective rule loading algorithm, which automatically composes an OWL2 RL ruleset depending on the input ontology. In our benchmarks, we found that this domain-based rule selection can significantly improve performance. Another body of work studies the extraction of a (smallest) module from a larger ontology, which still captures the meaning of a particular set of terms (e.g., yielding the same relevant entailments or query results) [15], [36], [43]. This kind of approach could be useful to support semantically-enhanced, rule-based service matching, by automatically extracting relevant ontology parts for service descriptions (see Section 6.2.2).

Yus et al. [62] analyzed whether currently available DL reasoners are deployable on Android devices. However, their evaluation is limited to classification, and does not consider OWL2 RL-based reasoners. They found that performance greatly depends on the engine and ontology size, with times ranging from 4s–1609s. Interestingly, they found a performance increase of ca. 30% between mobile devices only 1 year apart, which is a promising evolution. Nonetheless, Yus et al. [62] and Kazakov et al. [28] found orders of magnitude difference between PC and Android reasoning times. As future work, Kazakov et al. aim to study the reason behind this poor performance on smartphones. Yus et al. point to Android memory restrictions (and e.g., resulting garbage collections) being the main barrier to efficient performance, although further study needs to validate this claim.

Patton et al. [44] report that, due to the single-threaded nature of most reasoners, a near linear relation exists between consumed energy and computing time for OWL inferences on mobile systems. As such, energy usage estimates, based on reasoning times, could already be realistic. Regardless, future work involves recording detailed battery measurements.

8. Conclusion and Future Work

This paper presented the following contributions:

- **A selection of OWL2 RL subsets**, with the goal of optimizing reasoning performance on mobile systems. Orthogonally, these methods include OWL2 RL-conformant vs. non-conformant selections; and selections suiting “stable” (i.e., not subject to frequent and significant changes) vs. “volatile” ontologies. Our benchmarks showed that, depending on ontology volatility and need for conformity, these selections may greatly improve performance.

- **The MobiBench cross-platform, extensible mobile benchmark framework**, for evaluating mobile reasoning performance. Given a reasoning setup, including process flow, reasoning task, ruleset (if any) and ontology, developers can use MobiBench to benchmark reasoners on mobile platforms, and thus find the best system for the job. The large differences in performance between engines and scenarios, as observed in our benchmarks, clearly point towards the need for such a framework. To facilitate the developer’s job, the framework includes a Semantic Web layer, selection and pre-processing services, as well as automation and analysis tools. Further, we indicated the extensibility for each component, allowing developers to easily plug in new variants.

- **Mobile benchmarks**, which measure reasoning performance when materializing ontology inferences; focusing on the impact of different OWL2 RL ruleset selections, as well as the computational cost of best-performing OWL2 RL rulesets for particular scenarios and systems. We put these performance results side-by-side with the performance of 3 OWL2 DL reasoners. Depending on the concrete scenario, we found that OWL2 RL reasoning can be greatly optimized. Further, we showed the distinct utility of the semantic enhancement of service matching, with performance overhead depending on application constraints.

- **A study of the usefulness of OWL2 RL in mobile semantic reasoning**. By outfitting rule-based tasks, such as service matching, with an OWL2 RL ruleset, ontological knowledge can be leveraged to improve results. Service matching is a useful task in mobile settings, as it enables mobile apps to, e.g., identify services in smart environments [58]. As such, our work contributes to studying both the feasibility and utility of OWL2 RL on mobile systems.

Despite the presented work, as well as advancements reported in the state of the art, scalable mobile performance remains elusive. A huge gap still looms between PC and mobile reasoning times. Therefore, future work includes integrating additional optimization methods into MobiBench, such as utilizing rule templates for all rules. Optimizing and porting domain-specific rule selection, in light of its positive impact on performance, is also an avenue of future work. Similarly, we aim to deploy pre-processing solutions for n-ary rules directly on the mobile device, and compare their performance on an ontology corpus featuring large amounts of relevant n-ary assertions. Regarding service matching, we aim to represent user goals and services as complex class descriptions, which allows benchmarking service matching via OWL2 DL entailment. Measuring energy consumption, an important aspect for mobile systems, is also part of future work.

Our major focus in this paper was on materializing ontology inferences. Reasoning per query (via e.g., SLG) may also have its merits on mobile platforms, since it does not require a priori materialization. Studying its performance on mobile systems is considered a major avenue of future work. Finally, identifying additional OWL2 RL rule subsets for particular reasoning tasks (such as instance checking and realization) is also viewed as future work.

Acknowledgments

This research project is funded by a research grant from Bayer Healthcare.

References

[1] S. Ali and S. Kiefer, “microOR --- A Micro OWL DL Reasoner for Ambient Intelligent Devices,” in *Proceedings of the 4th International Conference on Advances in Grid and Pervasive Computing*, 2009, pp. 305–316.

[2] N. Ambroise, S. Boussonnie, and A. Eckmann, “A Smartphone Application for Chronic Disease Self-Management,” in *Proceedings of the 1st Conference on Mobile and Information Technologies in Medicine*, 2013.

[3] J. Angele *et al.*, “Web Rule Language (W3C Member Submission 2005),” 2005. [Online]. Available: http://www.w3.org/Submission/WRL/.

[4] Apache, “Apache Jena.” [Online]. Available: https://jena.apache.org/. [Accessed: 17-Mar-2017].

[5] J. Bak, M. Nowak, and C. Jedrzejek, “RuQAR: Reasoning Framework for OWL 2 RL Ontologies,” in *The Semantic Web: ESWC 2014 Satellite Events, Anissaras, Crete, Greece, May 25-29, 2014, Revised Selected Papers*, 2014, vol. 8798, pp. 195–198.

[6] C. Becker, “RDF Store Benchmarks with DBpedia comparing Virtuoso, SDB and Sesame,” 2008. [Online]. Available: http://wifo5-03.informatik.uni-mannheim.de/benchmarks-200801/.

[7] C. Becker and C. Bizer, “DBpedia Mobile: A Location-Enabled Linked Data Browser.,” in *LDOW*, 2008, vol. 369.

[8] C. Beeri and R. Ramakrishnan, “On the Power of Magic,” in *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1987, pp. 269–284.

[9] B. Bishop and S. Bojanov, “Implementing OWL 2 RL and OWL 2 QL Rule-Sets for OWLIM.,” in *OWLED*, 2011, vol. 796.

[10] B. Bishop and F. Fischer, “IRIS - Integrated Rule Inference System,” in *Proceedings of the 1st Workshop on Advancing Reasoning on the Web: Scalability and Commonsense*, 2008.

[11] C. Bizer and A. Schultz, “The berlin sparql benchmark,” *Int. J. Semant. Web Inf. Syst. Issue Scalability Perform. Semant. Web Syst.*, 2009.

[12] H. Boley, S. Tabet, and G. Wagner, “Design Rationale of RuleML: A Markup Language for Semantic Web Rules,” in *Proc. Semantic Web Working Symposium*, 2001, pp. 381–402.

[13] D. Calvanese *et al.*, “OWL2 Web Ontology Language Profiles (Second Edition),” 2012. [Online]. Available: http://www.w3.org/TR/owl2-profiles/#OWL_2_RL.

[14] W. Chen and D. S. Warren, “Towards Effective Evaluation of General Logic Programs,” in *The 12th ACM Symposium on Principles of Database Systems (PODS)*, 1993.

[15] B. Cuenca Grau, I. Horrocks, Y. Kazakov, and U. Sattler, “Modular Reuse of Ontologies: Theory and Practice,” *J. Artif. Intell. Res.*, vol. 31, pp. 273–318, 2008.

[16] R. Cyganiak and A. Jentzsch, “Linking Open Data cloud,” 2014. [Online]. Available: http://lod-cloud.net/versions/2014-08-30/lod-cloud.svg.

[17] R. U. Faruqui and W. MacCaull, “OwlOntDB: A Scalable Reasoning System for OWL 2 RL Ontologies with Large ABoxes,” *Found. Heal. Inf. Eng. Syst.*, vol. 7789, pp. 105–123, 2013.

[18] C. L. Forgy, “Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem,” *Artif. Intell.*, vol. 19, no. 1, pp. 17–37, 1982.

[19] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang, “HermiT: An OWL 2 Reasoner,” *J. Autom. Reason.*, vol. 53, no. 3, pp. 245–269, 2014.

[20] J. Gray, *The Benchmark Handbook for Database and Transaction Systems (2nd Ed.)*. Morgan Kaufmann, 1993.

[21] W. S. W. Group, “SPARQL 1.1 Overview (W3C Recommendation 21 March 2013),” 2013. [Online]. Available: http://www.w3.org/TR/sparql11-overview/.

[22] Y. Guo, Z. Pan, and J. Heflin, “LUBM: A benchmark for OWL knowledge base systems,” *Web Semant. Sci. Serv. Agents World Wide Web*, vol. 3, no. 2, pp. 158–182, 2005.

[23] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, “Maintaining Views Incrementally,” in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 1993, pp. 157–166.

[24] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, and S. Rudolph, “OWL 2 Web Ontology Language Primer (Second Edition),” 2012. [Online]. Available: http://www.w3.org/TR/owl2-primer/. [Accessed: 14-Apr-2015].

[25] A. Hogan and S. Decker, “On the Ostensibly Silent `W' in OWL 2 RL,” in *Proceedings of the 3rd International Conference on Web Reasoning and Rule Systems*, 2009, pp. 118–134.

[26] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, “SWRL: A Semantic Web Rule Language Combining OWL and RuleML (W3C Member Submission 21 May 2004),” 2004. [Online]. Available: http://www.w3.org/Submission/SWRL/.

[27] M. Karamfilova and B. Bishop, “SwiftOWLIM Reasoner,” 2011. [Online]. Available: https://confluence.ontotext.com/display/OWLIMv35/SwiftOWLIM+Reasoner#SwiftOWLIMReasoner-PerformanceOptimizationsinRDFSandOWLSupport.

[28] Y. Kazakov and P. Klinov, “Experimenting with ELK Reasoner on Android,” in *Proceedings of the 2nd International Workshop on OWL Reasoner Evaluation, Ulm, Germany, July 22, 2013*, 2013, pp. 68–74.

[29] Y. Kazakov, M. Krötzsch, and F. Simančík, “The Incredible ELK: From Polynomial Procedures to Efficient Reasoning with EL Ontologies,” *J. Autom. Reason.*, vol. 53, no. 1, pp. 1–61, 2014.

[30] C. Keller, R. Pöhland, S. Brunk, and T. Schlegel, “An Adaptive Semantic Mobile Application for Individual Touristic Exploration,” in *HCI (3)*, 2014, pp. 434–443.

[31] T. Kim, I. Park, S. J. Hyun, and D. Lee, “MiRE4OWL: Mobile Rule Engine for OWL,” in *Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, 2010, pp. 317–322.

[32] M. Klusch, M. Alam Khalid, P. Kapahnke, B. Fries, and M. Vasileski, “OWL-S Service Retrieval Test Collection (Version 4.0),” 2005. [Online]. Available: http://projects.semwebcentral.org/projects/owls-tc/.

[33] H. Knublauch, “OWL 2 RL in SPARQL using SPIN.” [Online]. Available: http://composing-the-semantic-web.blogspot.ca/2009/01/owl-2-rl-in-sparql-using-spin.html.

[34] H. Knublauch, “The TopBraid SPIN API,” 2014. [Online]. Available: http://topbraid.org/spin/api/.

[35] H. Knublauch, J. A. Hendler, and K. Idehen, “SPIN - Overview and Motivation (W3C Member Submission 22/02/2011),” 2011. [Online]. Available: http://www.w3.org/Submission/spin-overview/.

[36] B. Konev, C. Lutz, D. Walther, and F. Wolter, “Model-theoretic inseparability and modularity of description logic ontologies,” *Artif. Intell.*, vol. 203, pp. 66–103, 2013.

[37] M. Krötzsch, “The Not-So-Easy Task of Computing Class Subsumptions in OWL RL,” Springer, Berlin, Heidelberg, 2012, pp. 279–294.

[38] S. Liang, P. Fodor, H. Wan, and M. Kifer, “OpenRuleBench: An Analysis of the Performance of Rule Engines,” in *Proceedings of the 18th International Conference on World Wide Web*, 2009, pp. 601–610.

[39] N. Matentzoglu, S. Bail, and B. Parsia, “A Snapshot of the OWL Web,” in *The Semantic Web – ISWC 2013 – 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part I*, 2013, pp. 331–346.

[40] G. Meditskos and N. Bassiliades, “DLEJena: A Practical Forward-chaining OWL 2 RL Reasoner Combining Jena and Pellet,” *Web Semant.*, vol. 8, no. 1, pp. 89–94, Mar. 2010.

[41] B. Motik, I. Horrocks, and S. M. Kim, “Delta-reasoner: A Semantic Web Reasoner for an Intelligent Mobile Platform,” in *Proceedings of the 21st International Conference Companion on World Wide Web*, 2012, pp. 63–72.

[42] M. O’Connor and A. Das, “A Pair of OWL 2 RL Reasoners,” in *OWL: Experiences and Directions Workshop 2012*, 2012.

[43] J. Pathak, T. M. Johnson, and C. G. Chute, “Survey of modular ontology techniques and their applications in the biomedical domain.,” *Integr. Comput. Aided. Eng.*, vol. 16, no. 3, pp. 225–242, Aug. 2009.

[44] E. W. Patton and D. L. McGuinness, “A Power Consumption Benchmark for Reasoners on Mobile Devices,” in *13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014.*, 2014, vol. 8796, pp. 409–424.

[45] E. Puertas, M. L. Prieto, and M. De Buenaga, “Mobile

Application for Accessing Biomedical Information Using Linked Open Data," in *Proceedings of the 1st Conference on Mobile and Information Technologies in Medicine*, 2013.

[46] D. Reynolds, "OWL 2 RL in RIF (Second Edition)," 2013. [Online]. Available: http://www.w3.org/TR/rif-owl-rl/.

[47] V. Reynolds, M. Hausenblas, A. Polleres, M. Hauswirth, and V. Hegde, "Exploiting linked open data for mobile augmented reality," in *W3C Workshop: Augmented Reality on the Web*, 2010.

[48] M. Schneider and K. Mainzer, "A Conformance Test Suite for the OWL 2 RL RDF Rules Language and the OWL 2 RDF-Based Semantics," in *6th International Workshop on OWL: Experiences and Directions*, 2009.

[49] C. Seitz and R. Schönfelder, "Rule-Based OWL Reasoning for Specific Embedded Devices," in *10th International Semantic Web Conference, Bonn, Germany, Proceedings, Part II*, 2011, vol. 7032, pp. 237–252.

[50] A. Sinner and T. Kleemann, "KRHyper - In Your Pocket," in *Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings*, 2005, vol. 3632, pp. 452–457.

[51] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A Practical OWL-DL Reasoner," *Web Semant.*, vol. 5, no. 2, pp. 51–53, Jun. 2007.

[52] M. Smith, I. Horrocks, M. Krotzsch, and B. Glimm, "OWL 2 Web Ontology Language Conformance (Second Edition)," *W3C Recommendation*, 2012. [Online]. Available: http://www.w3.org/TR/owl2-test/.

[53] W. Tai, J. Keeney, and D. O'Sullivan, "Resource-constrained reasoning using a reasoner composition approach," *Semant. Web*, vol. 6, no. 1, pp. 35–59, 2015.

[54] The National Center for Biomedical Ontology, "BioPortal," 2016. [Online]. Available: http://bioportal.bioontology.org/. [Accessed: 17-Mar-2017].

[55] M. Wilson, A. Russell, D. A. Smith, A. Owens, and M. C. Schraefel, "mSpace Mobile: A Mobile Application for the Semantic Web," in *User Semantic Web Workshop, ISWC2005*, 2005.

[56] W. Van Woensel, "MobiBench Online Documentation," 2016. [Online]. Available: https://niche.cs.dal.ca/materials/mobi_bench/.

[57] W. Van Woensel, S. Casteleyn, E. Paret, and O. De Troyer, "Mobile Querying of Online Semantic Web Data for Context-Aware Applications," *IEEE Internet Comput. Spec. Issue (Semantics Locat. Serv.*, vol. 15, no. 6, pp. 32–39, 2011.

[58] W. Van Woensel, M. Gil, S. Casteleyn, E. Serral, and V. Pelechano, "Adapting the Obtrusiveness of Service Interactions in Dynamically Discovered Environments," in *Proceedings of the 9th International Conference on Mobile and Ubiquitous Systems*, 2012, pp. 250–262.

[59] W. Van Woensel, N. Al Haider, A. Ahmad, and S. S. R. Abidi, "A Cross-Platform Benchmark Framework for Mobile Semantic Web Reasoning Engines," in *13th International Semantic Web Conference, Riva del Garda, Italy. Proceedings, Part I*, 2014, pp. 389–408.

[60] W. Van Woensel, N. Al Haider, P. C. Roy, A. M. Ahmad, and S. S. Abidi, "A Comparison of Mobile Rule Engines for Reasoning on Semantic Web Based Health Data," in *2014 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2014)*, 2014, pp. 126–133.

[61] W. Van Woensel, P. C. Roy, S. Abidi, and S. S. Abidi, "A Mobile & Intelligent Patient Diary for Chronic Disease Self-Management," in *15th World Congress on Health and Biomedical Informatics*, 2015.

[62] R. Yus, C. Bobed, G. Esteban, F. Bobillo, and E. Mena, "Android goes Semantic: DL Reasoners on Smartphones," in *Proceedings of the 2nd International Workshop on OWL Reasoner Evaluation, Ulm, Germany*, 2013, pp. 46–52.

[63] S. Zander, C. Chiu, and G. Sageder, "A computational model for the integration of linked data in mobile augmented reality applications," in *Proceedings of the 8th International Conference on Semantic Systems*, 2012, pp. 133–140.

[64] S. Zander and B. Schandl, "A framework for context-driven RDF data replication on mobile devices," in *Proceedings of the 6th International Conference on Semantic Systems*, 2010, p. 22:1--22:5.

[65] C. Ziegler, "Semantic web recommender systems," in *In Proceedings of the Joint ICDE/EDBT Ph.D. Workshop 2004 (Heraklion*, 2004, pp. 78–89.

[66] "AndroJena." [Online]. Available: https://github.com/lencinhaus/androjena. [Accessed: 01-May-2017].

[67] "Apache Cordova." [Online]. Available: https://cordova.apache.org/.

[68] "Apache Jena Inference Support." [Online]. Available: https://jena.apache.org/documentation/inference/.

[69] "Appcelerator Titanium." [Online]. Available: http://www.appcelerator.com/mobile-app-development-products/.

[70] "Chrome DevTools." [Online]. Available: https://developer.chrome.com/devtools.

[71] "JFact." [Online]. Available: http://jfact.sourceforge.net/.

[72] "Manchester OWL Repository." [Online]. Available: http://mowlrepo.cs.manchester.ac.uk/datasets/mowlcorp/. [Accessed: 16-Jun-2016].

[73] "Nashorn JavaScript Engine." [Online]. Available: http://www.oracle.com/technetwork/articles/java/jf14-nashorn-2126515.html.

[74] "Oxford Ontology repository." [Online]. Available: http://www.cs.ox.ac.uk/isg/ontologies/. [Accessed: 16-Jun-2016].

[75] "RDFQuery." [Online]. Available: https://code.google.com/p/rdfquery/wiki/RdfPlugin.

[76] "RDFStore-JS." [Online]. Available: http://github.com/antoniogarrote/rdfstore-js.

[77] "W3C Forum Post on OWL2 RL test cases." [Online]. Available: http://lists.w3.org/Archives/Public/public-owl-dev/2010AprJun/0074.html.

[78] "Web Data Commons." [Online]. Available: http://webdatacommons.org/structureddata/.