On Serializable Incremental Semantic Reasoners

Carlos Bobed
IRISA/Université de Rennes 1, France
carlos.bobed-lisbona@irisa.fr

Fernando Bobillo
Eduardo Mena
I3A, University of Zaragoza, Spain
{fbobillo,emena}@unizar.es

Jeff Z. Pan
University of Aberdeen, UK
http://homepages.abdn.ac.uk/jeff.z.pan/pages/

ABSTRACT

This short paper motivates the need for incremental and serializable semantic reasoners. Two possible scenarios where semantic reasoners with these feature could be interesting are outlined, namely reasoning on mobile devices and managing dynamic knowledge.

CCS CONCEPTS

• **Human-centered computing → Ubiquitous and mobile computing**; • **Computing methodologies → Knowledge representation and reasoning**;

KEYWORDS

Knowledge management, Data dynamicity, Mobile computing

ACM Reference Format:
Carlos Bobed, Fernando Bobillo, Eduardo Mena, and Jeff Z. Pan. 2017. On Serializable Incremental Semantic Reasoners. In *K-CAP 2017: K-CAP 2017: Knowledge Capture Conference, December 4–6, 2017, Austin, TX, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3148011.3154462

1 INTRODUCTION

Ontologies have become a de-facto standard for knowledge representation as they improve knowledge sharing, reusing, maintenance, and decoupling from applications. *Semantic reasoners* are software implementations providing an automatic discovery of implicit knowledge that can be logically inferred from ontology axioms. Many reasoners for the standard OWL 2 ontology language and its profiles have been developed, such as *Fact++*, *JFact* or *TrOWL*.

A *persistent* semantic reasoner can save its internal state together with some precomputed inferences and reload it when presented with the same ontology. If a persistent reasoner receives as input a previously considered ontology, it will reuse previously computed calculations, avoiding the repetition of solving these tasks. Typically, storing the result of the ontology classification is interesting because most of the semantic reasoners start by computing it before solving any other reasoning task.

Going a step further, a *serializable* semantic reasoner can clone the data structures that represent its internal object state, obtaining two or more independent instances of the reasoner that can evolve in parallel. Serializable reasoners depend on the version of the reasoner: Small changes in the code of the reasoner could require changes in the serialization. For this reason, some reasoners such as *FaCT++* chose to store only the precomputed inferences, from which it is possible to retrieve the internal state of the reasoner independently of the version. Despite of this, the study of serializable reasoners could be interesting to reduce the execution time as much as possible.

On the other hand, a semantic reasoner is *incremental* if it can manage changes in the ontology without restarting the reasoning from scratch, that is, avoiding reclassifying the ontology. A typical scenario is the building stage step of the ontology, when the developer often realizes that the ontology is inconsistent and introduces some corrections to correct the problem, thus needing to reclassify a (usually) very similar ontology. In incremental reasoners, it is easier to manage the addition of new axioms than the update or deletion of existing axioms, the main reason is that a given axiom can be inferred from different sets of axioms. Actually, we will only assume that incremental reasoners are able to add new axioms.

A first scenario where serializable incremental reasoners might be helpful is semantic reasoning in mobile devices. In several real-world scenarios, the use of semantic reasoners can notably enhance mobile apps. Because of the typical limited computational capabilities of mobile devices but also for privacy reasons, we argue that in some cases it makes sense to use an external service to classify the ontology (or part of it) and return a serialized reasoner back. Then, the mobile device can deserialize it and use it locally (for example, to add private information).

Another scenario is the management of dynamic knowledge in ontologies. In situations where the knowledge is highly volatile, semantic reasoners are less appropriate because they assume monotonic reasoning and do not support an easy retraction of previous inferences. We argue that in some cases it makes sense to restart the reasoning but, instead of starting again from scratch, a serialized state of the reasoner can be a better starting point.

Unfortunately, to the best of our knowledge none of the existing semantic reasoners is serializable and incremental. *JFact* reasoner is serializable: It takes advantage of the Java mechanisms for serialization and is able to save a binary file. However, although incremental reasoning is partially implemented, it is not fully operative. *FaCT++* is claimed to be incremental and persistent, although not serializable. Indeed, it is able to save a text file with a representation of the ontology (with some changes, e.g., URIs are encoded as integers), the reasoner state, etc. Being persistent could be acceptable sometimes, but we have checked that incremental reasoning using a restored version of the reasoner does not always give the correct results. Finally, other reasoners, such as *CEL*, *ELK*, *Pellet*, and *Snorocket*, implement some kind of incremental reasoning but do not support serialization.

In the following we describe both scenarios (reasoning on mobile devices and managing dynamic knowledge) in depth to encourage reasoner developers to extend their systems to support them.

ACM ISBN 978-1-4503-5553-7/17/12…$15.00
https://doi.org/10.1145/3148011.3154462

2 MOBILE SEMANTIC REASONING

In our daily lives we typically use a significant number of mobile applications (*apps*), e.g., to receive information about the weather or the traffic. Enhancing such applications by using a semantic reasoner seems promising but also challenging (typically, there are limitations in terms of CPU power, memory, connectivity, etc.). There are three main ways to use a reasoner: external, local, or hybrid.

The first possibility is using an *external reasoner*, where the app relies on external servers which perform all the calculations. The most important advantage of this approach is that one can consider a server which is as powerful as required by the application. The main problem is that the semantic app needs to send a lot of data through the network; this can be a serious limitation in mobile computing environments where connectivity is often unreliable or even unavailable.

The second possibility is the use of a *local reasoner*, which can be native (implemented for a specific mobile device) or ported (to reuse existing semantic reasoners [2]). Now, ontologies are firstly classified by the reasoner, and then it is possible to answer queries (locally) over it. This is more appropriate when privacy is important or when the connectivity with the server can be faulty or nonexistent, as usual in mobile computing scenarios. As limitations, there is some evidence that reasoning time in ported reasoners is affordable in small or not very expressive ontologies [2]. One of the reasons is that optimization techniques are designed for desktop computers but not for mobile devices: for example, it is common to use more memory to save reasoning time, but mobile devices can have serious memory limitations.

A third possibility is following a *hybrid approach*, where the classification of the ontologies (or the materialization if the application only considers RDF triple stores) is computed using an external reasoner and query answering is performed locally. Here, there is only communication with the server once; then query answering can be performed several times by a local reasoner over the classified ontology. While the external reasoner can use a powerful server taking advantage of the implemented optimization techniques, user privacy can be compromised and an additional transmission time must be taken into account. We describe below this option in depth.

Selection of the best option. Given an ontology, we propose to build a simple *decision support module* to automatically determine which is the best option to use a semantic reasoner: external, local, or hybrid. The optimal (or maybe Pareto-optimal) choice depends on the application or, more specifically, on several factors such as the size and expressivity of the ontologies, the connectivity of the mobile device, privacy configuration, the particular reasoning task, etc. To this end, it is necessary to identify some metrics or features from which it is possible to take the final decision. There exist some preliminary works on the prediction of the cost in terms of time on desktop computers [5] or energy consumption on Android devices [4], but further efforts and new criteria are still needed.

The hybrid approach. We propose four different ways to implement it. *Firstly*, the server can send a classified ontology (with all the inferred subsumption relationships explicitly represented) back. *Secondly*, if the mobile device has a copy of the ontology, the server can send only a list of the inferred subsumption relationships and the axioms can be integrated on the mobile device. *Thirdly*, if the reasoner is serializable, the external server can send instead a copy of the reasoner. The local reasoner avoids the cost of loading the ontology, but at the cost of requiring that both devices (the server and the mobile) use the same reasoner and version. Both devices should also share a common serialization strategy (e.g., a Java virtual machine –in the server– does not serialize data in the same way as a Dalvik/ART virtual machine –in an Android device). And *fourthly*, if the mobile device has a copy of the ontology, the external server can provide a serialized version of the reasoner but not including the original ontology, which will be locally integrated. This requires some additional time to add the axioms but reduces the size of the transmission.

Note that in the second and fourth cases it is desirable to have an incremental reasoner, so the addition of the axioms produces an efficient update of the classification. While the serialized objects obtained in the third case can directly be reused in the local device once deserialized (if they share a serialization strategy), they have a large size. We find interesting to investigate whether it is really necessary to include the original ontology, or it is preferable sending everything but the original ontology, as in the fourth case, which can be integrated to the reasoner at the local device (if it contains a copy of the ontology). Perhaps a serializable reasoner should provide both options: *(i)* cloning to obtain a complete copy of the reasoner, and *(ii)* cloning to obtain a copy of the reasoner with the inferences but maybe not with the axioms in the original ontology.

In mobile computing scenarios we always need to consider the possibility of having limited connectivity. If this is the case, one may consider scenarios where the external server sends a smaller fragment rather than the complete one. Depending on the different ways to implement the hybrid approach, the external server could send: *(i)* a partially classified ontology (an ontology with some of the inferred subsumption relationships explicitly represented), *(ii)* a partial list of the inferred subsumption relationships, or *(iii)* a serialized version of a reasoner using a fragment of the ontology. The percentage of information to be sent can be decided according to the state of the network. The rest of the information is already available at the server, so it can send it to the mobile device later. Hopefully, there would be a better connectivity but, even if not, in the meanwhile the mobile device could have received something at least.

After fixing the size of the transmission, one can define several policies to decide which information to send, such as maximizing the size of transferred data, taking into account user preferences (the most usually accessed ontology elements), etc. In the ideal scenario, the ontology should be divided into independent *modules*, so the external reasoner could send the result of classifying one or several of them. Given the usual limitations on memory, the mobile device should make an appropriate management of the modules, keeping the most usual ones or those that are currently being used in the main memory, and storing the other ones in secondary memory.

Note that if the mobile device has a local copy of the original ontology, it can finish the classification by locally computing the remaining inferences. Otherwise, the mobile device would have to reason with incomplete information. Hence, reasoning would be incomplete (some correct inferences might be missing) but, since semantic reasoners implement monotonic reasoning, the results would be correct. This approximate reasoning is not the optimal option, but when there is no connectivity it is preferable than providing no result at all.

Experiments. Table 1 shows the typical size of serialized ontologies using *FaCT++* 1.6.4 through OWLAPI 3.5.5, and *JFact* 4.0.4

	JFact					*FaCT++*			
Ontology	SerOnt	SerRea	SerInf	SerT	ResT	SerOnt	SerInn	SerT	ResT
00004	0.99	2.33	101.34	1780.47	1.93	1.46	0.41	178.85	0.43
00035	TO	TO	TO	TO	TO	TO	TO	TO	TO
00347	4.38	12.51	102.45	409.29	2.17	8.02	3.86	137.62	0.66
00368	21.10	32.19	726.38	292.93	9.38	47.98	15.41	5.15	RE
00371	TO	TO	TO	TO	TO	ER	ER	ER	ER
00374	TO	TO	TO	TO	TO	ER	ER	ER	ER
00386	22.79	34.91	836.19	501.84	10.19	51.73	18.95	1436.83	RE
00390	21.25	32.46	732.41	352.03	7.12	48.23	16.26	6.64	RE
00398	TO	TO	TO	TO	TO	TO	TO	TO	TO
00400	23.66	36.11	878.77	810.91	9.13	ER	ER	ER	ER
00462	TO	TO	TO	TO	TO	TO	TO	TO	TO
00463	7.22	11.27	178.60	291.61	1.96	15.60	7.32	343.63	RE
00631	3.37	5.21	43.02	9.97	0.9	8.04	2.48	0.32	RE
00678	7.77	12.66	147.03	118.08	1.67	18.04	6.50	472.72	RE
00680	7.80	12.69	147.48	102.21	1.96	18.12	6.18	16.36	RE
00761	1.59	3.29	22.71	343.08	1.41	2.76	2.54	110.01	2.33
01c1c6df-2a	TO	TO	TO	TO	TO	TO	TO	TO	TO
16a2cd46	ER	ER	ER	ER	ER	TO	TO	TO	TO
290113a0	2.53	7.77	54.53	99.81	1.05	6.26	5.37	1.61	0.43
3ebf89a1	20.11	31.20	725.40	314.95	9.44	46.99	15.46	5.26	RE
42d22996	2.67	6.58	60.91	19.04	1.42	4.18	2.48	1.23	0.39
5043d1d7	2.60	7.02	ER	ER	ER	3.48	ER	ER	ER
53f9f8e8	0.72	1.97	ER	ER	ER	TO	TO	TO	TO
63153319	2.25	5.08	ER	ER	ER	4.04	ER	ER	ER
645a3ff8	0.57	1.26	ER	ER	ER	0.96	ER	ER	ER
7e1c9977	0.66	1.60	ER	ER	ER	0.99	ER	ER	ER
cbe2e729	TO	TO	TO	TO	TO	ER	ER	ER	ER
cell	TO	TO	TO	TO	TO	TO	TO	TO	TO
d0e20d33	2.53	7.78	76.69	121.95	1.14	6.26	6.09	11.49	0.49
d5c7f91d	3.86	11.84	134.21	280.88	1.58	9.27	9.23	3.90	0.76
e5c03a5b	1.34	2.75	74.26	1658.40	1.7	1.97	1.22	310.27	2.42
ebf8d261	9.73	13.81	48.47	42.37	1.07	13.50	4.48	0.84	0.38
FMA	TO	TO	TO	TO	TO	TO	TO	TO	TO
GALEN-F	3.52	ER	ER	ER	ER	7.01	ER	ER	ER
GALEN-H	0.68	ER	ER	ER	ER	1.27	ER	ER	ER
teleost	0.32	0.78	5.79	2.86	0.33	0.76	1.02	0.65	0.08

Table 1: Serialization using *JFact* and *Fact++*.

through OWLAPI 4.2.7. The experiments were carried out in an MSI GE620 laptop (i7-2670QM CPU, 8GB RAM). *FaCT++* produces text files encoding the state of the reasoner that require some processing at the local file to create an object from this configuration file, whereas *JFact* produces serialized instances of a *JFact* reasoner in a binary format, but the obtained files usually have a very large file because they include the ontology. We considered 36 OWL 2 DL large ontologies (each of them having more than 5000 logical axioms) from the ORE 2013 dataset [3]. For each ontology, we created an instance of a reasoner and tried to classify the ontology. In case of success we serialized the reasoners and measured the size of the serializations.

For both reasoners we show the size in MB of the serialized ontology (*SerOnt)*, the time in seconds needed to obtain the serializations (*SerT*), and the time in seconds needed to restore the reasoner state from the serializations (*ResT*). For JFact we additionally show the sizes in MB of the serializations of the reasoner after its creation (*SerRea*) and after computing the classification (*SerInf*). For Fact++ we additionally show the size in MB of the serialized inner state of the reasoner (*SerInn*). Some found problems are also reported: *ER* indicates a runtime error, *TO* shows a timeout (i.e., the ontology classification was not completed after 120 minutes), and *RE* indicates an error while restoring the reasoner state from the serialization. For space limitations, ontology names are shortened.

We observe that *Fact ++*'s serialization requires significantly less space than *JFact*'s serialization after computing the classification, seeming the most reasonable approach to minimize data transmission. Surprisingly, the time needed to restore the ontologies is usually smaller in *Fact ++* too. Restoring time is, as expected, very low in comparison with the inference time, which is coherent with our proposal.

3 MANAGING VOLATILE KNOWLEDGE

A significant number of ontology-based applications require dealing with knowledge which is able to change very often. Indeed, data streams management is a popular topic. Attributes that could change anytime are common in many environments, e.g., availability of a taxi cab, occupants of a vehicle, or the *location* of mobile devices (which changes also very frequently and it is a key information for many apps). The imminent realization of the Internet of Things would increment the volume of this kind of dynamic data.

Managing dynamic knowledge in ontologies is challenging. If, for example, the definition of the class of available taxis depends on the number of occupants, the constant changes of the occupants require to reclassify the ontology continuously, so the performance of the system would dramatically decrease. Another problem is that the usual monotonic reasoning is no longer correct: When an axiom is removed, the consequences inferred from it might be invalid.

Most of the previous effort on reasoning with dynamic knowledge is restricted to RDF triples (see [1, Related Work]); these approaches usually materialize the ontologies by storing the inferred consequences of every axiom. While RDF might be enough for many applications, we argue that there exist applications which require a more expressive semantic language. Rather than keeping track of the consequences of every axiom to remove them, one idea is to separate the changeable and the unchangeable knowledge, and to empty all the changeable consequences when deleting an axiom.

In a previous work, we differentiate between static properties and volatile properties [1]. The values of a static property do not change or remain unchanged long enough to consider them static, but the values of a volatile property are expected to change anytime and frequently. However, [1] assumes a different semantics (it stores ontology axioms in databases and relies on Closed World Assumption) and gives priority to the static knowledge (volatile knowledge is only accepted if is consistent with the previous static information).

Our objective now is to give the same answers than any standard semantic reasoner. In the following we will propose some techniques to reason with ontologies including volatile properties. The main idea is to consider a stored version of the static part of the ontology classified together with a current version of the ontology including both the static and volatile parts. This way, every time an axiom is deleted, we can empty the consequences of the volatile properties by restarting the volatile part. Our hypothesis is that, in some cases (depending on the number of volatile properties and the update rate), it is more efficient to restart the current ontology from a snapshot than keeping track of the inferred axioms and removing them.

Partition of the ontology. Let O be an ontology with volatile properties; the status of O at a given point in time will be denoted as O_t. Static properties and volatile properties will be denoted as r_s and r_v, respectively. The sets of static and volatile properties will be denoted as S_{R_s} and S_{R_v}, respectively. One can assume that the ontology developer is responsible of annotating the properties as volatile or static during the ontology development, or we can try an automatic detection of the volatile properties [6], but this is challenging as the fact that some property value has not changed for some time does not imply that it will never change in the future.

Before solving any reasoning task, some preprocessing is needed. In particular, we split the ontology into two parts: a static one and a volatile one. The idea is that the static part is unchangeable and only needs to be classified once, but the volatile part of the ontology is changeable and requires some additional effort every time that there is a change in the value of a volatile property. Hence, we define the disjoint union $O = O_s \uplus O_v$, where $O_v = \{r_v(o_1,o_2) \mid r_v \in \mathsf{S}_R\}$ and $O_s = O \setminus O_v$. In particular, note that $O_s \cap O_v = \emptyset$. O_v can change over time, O_s cannot. The classified ontology, denoted as O_s^{class}, is reused for being unchangeable, so we make a copy of the reasoner at this point, which is possible if it is serializable. Finally, we obtain the classified ontology at the current time, denoted O_t^{class}, by classifying $O_s^{\text{class}} \cup O_v$. We use O_t^{class} to manage changes in volatile properties and perform inferences.

Adding new role assertions. The first possible change in the ontology is the addition of a new role assertion involving a volatile property. For each new role assertion $\tau = r_v(o_1,o_2)$ we (i) add τ to O_v, and (ii) recompute O_t^{class}, by classifying $O_t^{\text{class}} \cup \{\tau\}$. If the reasoner supports the incremental addition of axioms, our hypothesis is that this should be very fast. An alternative is processing a batch of n new role assertions, saving $n-1$ classifications, if no query is submitted to the system in the meanwhile. The case of concrete role assertions $\tau = r_v(o,v)$ is similar.

Removing new role assertions. The other possible change in the ontology is the removal of a new role assertion involving a volatile property. For each new role assertion $\tau = r_v(o_1,o_2)$ we (i) remove τ from O_v, and (ii) recompute O_t^{class} by classifying $O_s^{\text{class}} \cup O_v$. Now the usefulness of having saved a copy of O_s^{class} becomes apparent. The hypothesis is that since the static part is already classified, updating the concept hierarchy might be faster than computing the classification from scratch or than finding and removing the inferred consequences of τ. Note that we do not require the reasoner to be able to remove axioms incrementally. Again, the case of concrete role assertions $\tau = r_v(o,v)$ is similar.

Solving reasoning tasks. Queries submitted to the system are answered using O_t^{class}. Since it is always classified, the answers are computed faster (actually, almost all the semantic reasoners start by classifying the ontology before answering queries). The correctness of our approach can be easily proven, as shown next.

Theorem 3.1. *Given an ontology O_t at any point of time, the result of classifying O_t is equivalent to O_t^{class}.*

Proof. Trivial from the fact that we compute O_t correctly. If a new role assertion τ is added to the ontology, we essentially add it to O_v and to O_t^{class}. If a role assertion τ is removed from the ontology, we remove it from O_v and recompute O_t^{class} from $O_s^{\text{class}} \cup O_v$. □

Y. Ren et al. [7] show empirically that when the update rate is high, re-computation from scratch is preferable than incremental reasoning (15% update is the turning point). Our approach should outperform the cost of re-computation from scratch since we do not reclassify the static part of the ontology.

Table 2 compares the tasks required by our proposal and by a naïve approach. We can see that the query answering step is equivalent in both cases. The addition of axioms seems similar but in our approach we could save the cost of keeping track of the inferences that can be deduced from a given axiom, as we would not use incremental reasoning to remove axioms. In our approach, the initialization is more

Task	Our proposal	Naïve approach
Initialization	Split O_s and O_v O_s^{class} = classify(O_s) Save a copy of O_s^{class} $O_t = O_s^{\text{class}} \cup O_v$ O_t^{class} = classify(O_t)	O_t^{class} = classify(O)
Add	Add axiom to O_t^{class} Update incrementally O_t^{class}	Add axiom to O_t^{class} Update incrementally O_t^{class}
Remove	Remove axiom from O_v $O_t = O_s^{\text{class}} \cup O_v$ O_t^{class} = classify(O_t)	Remove axiom from O_t^{class} Update incrementally O_t^{class}
Query	Answer query over O_t^{class}	Answer query over O_t^{class}

Table 2: Tasks in our proposal and in a naïve approach

expensive but sometimes it could be compensated with the reduction in the cost of removing axioms: Reclassifying $O_s^{\text{class}} \cup O_v$ taking into account that O_s^{class} is already classified could be faster than removing all the inferences computed thanks to the removed axiom.

Implementation. Rather than developing a new reasoner from scratch, we propose building a *metareasoner* taking advantage of the plethora of highly optimized ontology reasoners that are available. Our Java prototype would be able to use any ontology reasoner that satisfies the following requisites: *(i)* it is accessible via the OWL API, *(ii)* it is *incremental*, and *(iii)* it is *serializable*. Being serializable is needed to save a copy of the data structures representing O_s^{class} and reuse them efficiently; being incremental is necessary from a practical point of view: If the reasoner needs to start the reasoning from scratch after every change, the approach is not viable. Unfortunately, to the best of our knowledge, none of the existing semantic reasoners satisfies all these restrictions yet.

ACKNOWLEDGMENTS

We were funded by projects TIN2013-46238-C4-4-R and TIN2016-78011-C4-3-R, and DGA-FSE. We would like to thank Jesús Bermúdez, Ignazio Palmisano, and Dmitry Tsarkov for some helpful discussions.

REFERENCES

[1] Carlos Bobed, Fernando Bobillo, Sergio Ilarri, and Eduardo Mena. 2014. Answering Continuous Description Logic Queries: Managing Static and Volatile Knowledge in Ontologies. *International Journal on Semantic Web and Information Systems* 10, 3 (2014), 1–44.

[2] Carlos Bobed, Roberto Yus, Fernando Bobillo, and Eduardo Mena. 2015. Semantic Reasoning on Mobile Devices: Do Androids Dream of Efficient Reasoners? *Journal of Web Semantics* 35, 4 (2015), 167–183.

[3] Rafael S. Gonçalves, Samantha Bail, Ernesto Jiménez-Ruiz, Nicolas Matentzoglu, Bijan Parsia, Birte Glimm, and Yevgeny Kazakov. 2013. OWL Reasoner Evaluation (ORE) Workshop 2013 Results: Short Report. In *Procs. of the 2nd Int. Workshop on OWL Reasoner Evaluation (ORE 2013)*, Vol. 1015. CEUR Workshop Proceedings, 1–18.

[4] Isa Guclu, Yuan-Fang Li, Jeff Z. Pan, and Martin J. Kollingbaum. 2016. Predicting Energy Consumption of Ontology Reasoning over Mobile Devices. In *Proceedings of the 15th International Semantic Web Conference (ISWC 2016), Part I (Lecture Notes in Computer Science)*, Vol. 9981. 289–304.

[5] Yong-Bin Kang, Jeff Z. Pan, Shonali Krishnaswamy, Wudhichart Sawangphol, and Yuan-Fang Li. 2014. How Long Will It Take? Accurate Prediction of Ontology Reasoning Performance. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI 2014)*. AAAI Press, 80–86.

[6] Yuan Ren and Jeff Z. Pan. 2011. Optimising Ontology Stream Reasoning with Truth Maintenance System. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management (CIKM 2011)*. 831–836.

[7] Yuan Ren, Jeff Z. Pan, Isa Guclu, and Martin J. Kollingbaum. 2016. A Combined Approach to Incremental Reasoning for EL Ontologies. In *Proceedings of the 10th International Conference on Web Reasoning and Rule Systems (RR 2016) (Lecture Notes in Computer Science)*, Vol. 9898. Springer, 167–183.