An Automatically Composable OWL Reasoner for Resource Constrained Devices

Wei Tai, Rob Brennan, John Keeney, Declan O'Sullivan
Knowledge and Data Engineering Group, School of Computer Science & Statistics, Trinity College Dublin, Dublin 2, Ireland
{TaiW, Rob.Brennan, John.Keeney, Declan.OSullivan}@cs.tcd.ie

Abstract

Centralized semantic sensor network systems gradually show performance degradation as the scale of the sensor network increases. Thus systems based on distributed approaches with local, autonomous management features are urgently required. In order to achieve local autonomy, it is necessary to push semantics towards the edge of the sensor network, but this is hampered by the lack of availability of lightweight ontology processing and reasoning technologies that are cognisant of the limited resources available in sensor network nodes. This paper proposes an approach to dynamically and automatically compose an ontology reasoner to provide only the level of OWL reasoning required for a given application. A design and prototype implementation are presented, with initial evaluations confirming that this approach saves memory without loss of reasoning ability, which facilitates OWL reasoning on the resource constrained devices typical in sensor networks.

1. Introduction

Scalability and dynamic adaption are key capabilities for sensor networks to function effectively. The development of the Semantic Sensor Web brings the additional promise of greater reusability and flexibility for both sensor data and sensors themselves. However the scale of these new networks, both in terms of numbers of devices and the volume of data generated, necessitate distributed approaches to both data services and management that emphasise local autonomy, for example P2P approaches or the new governance and processing models developed in the context of autonomic communications [1]. In autonomic communications systems local autonomy is directed towards maintaining system-wide goals rather than relying on direct, centralised control or decision-making. Hence it is our contention that as the network expands that the locations at which application level processing should take place must also expand, in some cases right to the edge of the sensor network itself. The corollary of this is that the widespread penetration of declarative semantics envisioned by the semantic sensor network should be accompanied by a spread of reasoning capabilities throughout the network. This is the best way to maximise the utility and application of these semantics while also enabling new capabilities at edge (or near-edge) nodes. Such reasoners will have to be appropriately dimensioned for the computational resources available at the edge of the network.

In this paper we describe a composable rule-based OWL reasoner that can be automatically composed to be just-fit in terms of the OWL constructs of the given ontology. Unnecessary rules are then not loaded into reasoner. A preliminary evaluation shows this approach reduces resource consumption (especially memory) for OWL reasoning without loss of reasoning ability. It differs from earlier work because customisation of the reasoner is performed automatically by analyzing the semantics of the actual ontology to be reasoned and selecting required reasoning modules rather than hand-crafting one-off customized reasoner implementations by experts.

This automatic composable reasoner is currently being implemented in a distributed wireless sensor network management system, where sensors are organized in clusters with each cluster managed by a cluster head. The automatic composable reasoner is employed in this system for root cause identification and source localization: to identify from network fault notifications their root causes and the ultimate source of failures (typically failures in one node cause a cascade of fault notifications from other nodes). In our prototype sensor network management system, manager functions are resident in both gateway

(standard PC) nodes and cluster heads (Sun SPOTs[1]). The Sun SPOTs, while still an embedded platform, are high-end devices supporting Java-based development with greater computational resources than the low power nodes that make up the majority of the wireless sensor network. It is these slightly more general purpose nodes that are the ultimate target of our current research, rather than the lowest power devices. As shown in section 5 this approach reduces the memory footprint without reducing the reasoning ability, which seems to be a promising approach to push intelligence down to resource constrained devices.

In this paper, section 2 gives a short introduction to related work on OWL reasoners, their categorization as well as how they modularize and re-compose their reasoning abilities. Section 3 includes the principles of this approach, a description of the prototype components and a detailed description of the rule set used. The implementation and the preliminary evaluation are in section 4 and section 5 respectively. Section 6 draws conclusions and discusses future work.

2. Related Work

In this section three bodies of previous research related to our aim are discussed: a brief survey of OWL reasoner implementation approaches; a discussion of the composability of existing rule-based entailment reasoners; and a brief overview of previous work on reasoning efficiency.

2.1. Current OWL Reasoner Implementations

A reasoner is defined as a system that "allows one to infer implicitly represented knowledge from the knowledge that is explicitly contained in a knowledge base" [2]. For an OWL reasoner the knowledge base is a set of OWL ontologies. There are a wide range of OWL reasoners in modern knowledge-based systems. Each implementation has its own functional and non-functional trade-offs, including: semantic expressiveness, computational complexity, memory footprint and processor load. We made a survey of a set of 28 modern OWL reasoners and categorized them into five categories according to their internal reasoning algorithms. The commonalities between reasoner implementations enabled us to direct our research based on types of reasoners rather than a specific implementation. The categories are as follows:

Description Logic (DL) Tableaux-based reasoners (e.g. Pellet [3]), perform OWL reasoning by translating OWL into DL and then reduce OWL reasoning to DL satisfiability checking. *OWL entailment rule-based reasoners* (e.g. Jena [4]) compute OWL entailment by matching OWL entailment rules in a rule engine against OWL ontologies either in forward chaining or backward chaining, or hybrid style. *First order logic (or its subset such as Prolog or Datalog) prover-based reasoners* (e.g. Hoolet [5], KAON2 [6]) transform OWL ontology into first order logic (or its subset) formulae and then delegate OWL reasoning to first order logic theorem prover (or engines of its subset). *F-logic-based reasoners* (e.g. FOWL [7]) map OWL ontology to f-logic and perform OWL reasoning using an f-logic engine. *SQL engine-based reasoners* (e.g. Oracle's OWLPrime [8]) model OWL entailment rules using SQL statements and thus OWL reasoning turns out to be evaluating SQL statements over OWL ontologies storing in databases.

OWL Entailment rule-based OWL reasoners are fast, small in memory footprint, are found to be easy to modularize and re-compose to a fine granularity based on the set of OWL entailment rules (See section 3). In addition, the use of a rule engine offers applications intrinsic ability not only to reason over OWL ontologies but also to process application specific rules, which is important for many applications, e.g. sensor network management. For these reasons our initial research has focused on automatic composability of entailment rule-based reasoners. For simplification, we use "rule" and "OWL entailment rule" interchangeably in the following text.

2.2. Reasoning Composition Approaches

Some previous work has been done on the problem of reasoning composition. A simple approach of performing reasoning composition is to provide pre-defined reasoning levels. For example, Jena provides three built-in reasoning levels, i.e. OWL, OWL Mini, and OWL Micro, which allow users to select through its API. Some rule-based reasoning systems even allow users to freely select rules from their rule set to construct their own reasoning level, e.g. the Oracle's OWLPrime. Jena uses plain text encoded rule files which potentially allow users to compose their own reasoning level. These approaches are mostly aiming at providing more flexibility rather than to improve reasoning performance or to lower resource consumption. Indeed, approaches given below are more focused at the later aim.

Meditskos and Bassiliades introduced in [9] the Incremental Loading of Rules and Incremental Loading of Triples methodology. It provides a type of algorithm level re-composition. OWL entailment rules are partitioned into subsets according to their related expressivity and the OWL ontology to be reasoned is

[1] http://www.sunspotworld.com/

also divided into portions containing a predefined number of triples. Then rule subsets are circularly loaded into the reasoner which reasons over the ontology as it is incrementally loaded. This approach improves the memory consumption (maximum memory) of OWL reasoning despite the extra memory expended applying rule subsets in a circular mode. However, it performs no semantic analysis on the ontology, thus unused rules even though partitioned into subsets are still loaded into the rule engine, resulting in unnecessary memory consumption.

Amir et al proposed in [10] a partition-based logical reasoning. It decomposes logical theories into signature-overlapped *partitions* and interconnects partitions into a tree-shaped *intersection graph* with partitions as vertices and arcs labelled by the common signature shared by adjacent two vertices. Consequence findings (e.g. resolution) are performed concurrently for all partitions once a query is received and logical consequences of each partition are propagated to the next adjacent partition toward the vertex whose signature covers the query formula. The query succeeds if it is finally proved in that vertex. This approach uses queries to guide selective consequence finding hence reducing the resolution search space and improves the reasoning efficiency.

In [11] Grau et al propose an approach which uses E-Connections [14] to partition OWL knowledge base into several modelling and logically disjoint sub-domains, and each sub-domain is modelled as a separate component ontology. *Link properties* are used to relate a component ontology with another. This approach allows the reasoner to load and reason over only the relevant portion, (i.e. E-connected component ontologies in a transitive closure under *link properties*,) of the OWL knowledge base, and leaves irrelevant component ontologies unloaded.

Guo et al propose in [13] a graph-based approach to partition large OWL ABoxes to allow specific reasoning performed on each partition and results are combined to a complete answer. The rationale behind this is to put specified assertions in the antecedent of an inference rule into the same partition (refer to [18] for all rules). A set of *Chunk Rules* is designed (based on the rule set given above) to construct a (directed) *Chunk Graph* for a given OWL knowledge base. Vertices are disjoint chunks of triples, and arcs from a chunk to another means a partition containing the first chunk must contain the second. Partitions are constructed by identifying all connected component. This approach helps reasoners to overcome memory limitation for knowledge bases with large ABoxes.

Fokoue et al in [19] propose an approach to reduce the size of real world ABoxes by aggregating similar individuals and assertions to build *summary ABoxes*. Meanwhile an efficient filtering technique is also proposed to perform ABox reasoning for specific reasoning tasks (e.g. consistency checking). Indeed it segments the summary ABox into partitions isolating only relevant portions of consistency checking. The summary technique and the filtering technique are proved to have large reduction from both space and time perspectives for consistency checking.

Composition approach proposed in this paper analyzes the expressivity of given OWL ontology and then loads only rules related to OWL constructs present in the ontology to be reasoned. This differs from some previous approaches which aim to partition ontologies rather than the reasoning rule set. A prototype implementation of this approach shows it reduces the memory footprint (see section 5), which shows promises for resource constrained devices.

3. Technical Approaches

The fundamental idea of the composable OWL reasoner proposed in this paper is to automatically compose the its rule set according to the present OWL constructs of the ontology to be reasoned. This includes the removal of rules irrelevant to the reasoning of the given ontology, while keeping just those required. A consequence of this approach is the reduction of the rule set size, leading to a reduction of execution time and memory consumption for RETE network building as well as ontology reasoning. This section presents the technical approaches we employed in our prototype as well as its design.

3.1. Candidate Rules

Candidate rules are those rules waiting for selecting. Since our prototype is designed mainly for proof of concept, the expressivity it supports is selected to cover only an OWL subset that can be easily expressed as rules (based on the pD* interpretation [12]). Candidate rules of this prototype are partly from the RDF(S) rule set [15] (7 rules), partly from the pD* rule set [12] (20 rules) and for the rest (9 rules) authored to complement the OWL constructs that are not covered by pD* entailment rules. Rules about number restrictions (i.e. owl:cardinality, owl:minCardinality, owl:maxCardinality), datatypes (e.g. lg, gl, rdf2, rdfs1 and rdfs13 from [15]), and some other rules which can cause the explosion of the size of ontology as well as inefficient entailment process by naïve application (e.g. se1, se2, rdf1, rdfs4a, rdfs4b, rdfs6, rdfs9 and rdfs10 from [15]), are excluded. For example, the naïve application of se1 and se2 leads to an infinite loop and produces redundant derivations of equivalent triples,

and the rule rdf1 asserts any predicate to be of type rdf:Property, which is optional in OWL semantics.

The soundness proof for RDFS entailment rules and pD* entailment rules can be found in [15] and [12] respectively. The soundness proof for the rules designed by ourselves is not listed here due to space limitations but it can be found on the web[2].

3.2. Rule Selection against OWL Constructs

Generally rule selection is a process of constructing a rule closure for rules selected according to the OWL constructs present in the ontology. First an initial rule set needs to be selected to start the rule closure calculation. In order to determine whether a rule needs to be selected, the selection process compares OWL constructs appearing in those terms from the left-hand side (l.h.s.) of the rule with OWL constructs present in the ontology to be reasoned: it is selected if all OWL constructs appearing in its l.h.s. terms also present in the OWL ontology. For example, the rule rdfp1 (given in Figure 1) has owl:FunctionalProperty in its l.h.s. terms, then it is loaded for the reasoning of an ontology if the ontology contains owl:FunctionalProperty as well. By doing this an initial rule set is selected.

```
(?p rdf:type owl:FunctionalProperty)
(?u ?p ?v)
(?u ?p ?w)
→
(?w owl:sameAs ?v)
```

Figure 1. The OWL entailment rule rdfp1

However, the initial rule set is not as yet a complete rule set for the reasoning of the given ontology as the firing of some selected rules may add new constructs that are not present in the original ontology, e.g. the owl:sameAs introduced by the rdfp1 may not appear in the original ontology, so rules to handle newly added constructs may not be present. To cope with this, a rule closure is calculated by considering OWL constructs appearing in right-hand side (r.h.s.) terms of the initial rule set as (potential) constructs of the OWL ontology and then selecting extra rules according to the criteria given above. In the above example, we suppose the rdfp1 introduces the owl:sameAs which is not contained in the original ontology so the rules rdfp11a, rdfp11b are also selected to handle any inserted owl:sameAs statements. This process iterates until no more new rules can be added. The selected rule set is then a rule closure for the given ontology and contains all required rules from the candidate rule set.

2 https://www.cs.tcd.ie/~taiw/interpretations/pDStarplus.doc

4. Design and Implementation

In this section the design and implementation of the prototype is given.

4.1. Components Description

The prototype (as illustrated in Figure 2) consists of four main components: the Construct Analyzer, the Rule Selector, a general-purpose rule engine and the Ontology-to-Facts Translator.

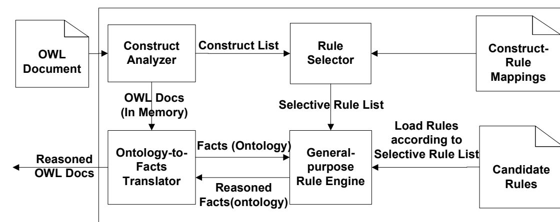

Figure 2. Composable reasoner components

The Construct Analyzer determines the OWL constructs included by the input ontology. Basically this is done by testing the existence of all OWL constructs over the loaded ontology through Jena. Successful OWL constructs are then those explicitly present in the ontology and they are passed to the Rule Selector as a list.

```
<InfRule name = "rdfp 16">
   <Level>OWL-LITE</Level>
   <FeaturedConstruct>http://www.w3.org/2002/07/owl#allValuesFrom</FeaturedConstruct>
</InfRule>

<InfRule name = "complement 2 disjoint">
   <Level>OWL-DL</Level>
   <FeaturedConstruct>http://www.w3.org/2002/07/owl#complementOf</FeaturedConstruct>
   <CascadingConstruct> http://www.w3.org/2002/07/owl#disjointWith</CascadingConstruct>
</InfRule>
```

Figure 3. A snippet of the construct rule mappings

The Rule Selector gives a selective rules list following the approach described in section 3.2. However, to avoid on-the-fly analysis of rules for OWL constructs, a XML document (Construct Rule Mappings, as shown in Figure 3) is constructed beforehand describing rule information required by Rule Selector, including name, constructs present in the l.h.s (tagged as FeaturedConstruct), constructs that will be added into ontology (tagged as CascadingConstruct), as well as level of reasoning it performs (i.e. RDFS, OWL-DL or OWL-LITE). This saves processing time and memory.

The actual ontology reasoning (i.e. rule evaluation) is performed in the Drools[3] general-purpose rule engine. It is selected in the first place because the previous experience with it in our group. Rules are

3 http://www.jboss.org/drools/

loaded according to the selective rule list given by the Rule Selector. Two different selective rule loading modes (i.e. additive and subtractive) are designed corresponding to the two rule loading modes of Drools. For the *additive* mode, rules are loaded from Drools rule source files and then compiled on-the-fly into RETE network in memory. This mode does not load unselected rules, and is best for situations where only a small subset of a large rule set is selected. For the *subtractive* mode, a pre-serialized RETE network containing the complete candidate rules set is loaded into memory and then non-selected rules are removed according to the selective rule list. This mode is best for situations where the selective rule set consists of the majority of the candidate rule set. Both modes are evaluated in this paper.

The Ontology-to-Facts Translator performs the bi-directional translation between the OWL ontology and Drools facts.

4.2. Implementation

The prototype was implemented using Java 1.6.0. The Drools rule engine v4.0.7 was used as the general-purpose rule engine, and Jena 2 was used for ontology access and management.

```
rule "rdfs3"
  when
    Triple(p:subject, predicate == http://www.w3.org/2000/01/rdf-schema#domain, u:object)
    Triple(v:subject, predicate == p)
    not (Triple(subject == v, predicate == http://www.w3.org/1999/02/22-rdf-syntax-ns#type,
          object == u))
  then
    insert(new Triple(v, http://www.w3.org/1999/02/22-rdf-syntax-ns#type, u));
```

Figure 4. A candidate rule in drools rule language

Components given by Figure 2 were implemented as separate Java classes with the FCRuleEntailmentReasoner as their coordinator. Candidate rules were kept in two different formats for the two selective rule loading modes. For additive mode candidate rules were loaded as Drools source files from rules.drl (as shown in Figure 4). For subtractive mode all candidate rules were kept in a pre-compiled rule base (rules.pkg) and loaded into Drools in whole with non-selected rules removed before reasoning.

A configuration class was implemented to allow configuration on the behaviour of the reasoner. Users can configure the location of rule source, language level for rule selection (including RDFS, OWL-Lite and OWL-DL), ontology location, rule loading mode, composability switch, etc.

5. A Preliminary Evaluation

This section describes a preliminary evaluation study of the execution time and memory consumption of our prototype.

5.1. Environment and Methodology

This evaluation was carried out on a desktop PC with Intel Core 2 CPU 6600 at 2.4GHz, 3.25GB RAM and Windows XP professional version 2002 SP2. This prototype was developed and evaluated in Eclipse 3.3.2 with JDK version 1.6.0 build 06 with –Xms32M and –Xmx32M. The Drools engine was configured to use stateless sessions.

Memory cost and execution time were measured for both a complete automatic composable reasoning process and its sub-steps, i.e. RETE network building, and ontology reasoning (i.e. rule evaluation). These tests were performed to prove our initial idea and also to identify performance bottleneck of this prototype. All tests were carried out on the prototype configured in 4 different combinations in terms of rule loading mode and composability: Additive On (composable reasoning in additive loading mode), Additive Off (non-composable reasoning in additive loading mode), similarly, Subtractive On and Subtractive Off.

Execution time and memory cost were tested using the built in Java time and memory methods. Each measurement was the average of 10 independent executions with percentage error ((standard deviation/mean)*100%) calculated. The Java garbage collector was invoked at least 10 times before each memory measurement to minimise the impact of previous memory allocations (the low percentage error observed in the measured values showed that this method was effective).

5.2. Ontology Selection

Five ontologies (as listed in Table 1 below) were selected for evaluation. The selection was motivated by the fact that they are of small or moderate sizes and different expressivities, which to some extent can simulate the ontology used for sensor network management (in many cases ontologies used for sensor network management are of low expressivity and small size). In addition, they were commonly accepted as examples for well known ontology tools such as Protégé[9], etc, which to a large extent avoided errors.

There are many other test suites for evaluating reasoners. For example, the OWL Test Case [16]

[9] http://protege.stanford.edu/

provides nine sets of test cases to evaluate different aspects of an OWL reasoner. However, it concentrates more on functional evaluation rather than performance evaluation. The LUBM are widely used for evaluating the query answering performance of OWL reasoners. It is not used for evaluating this approach as root cause identification is more subsumption checking-based rather than query answering-based. However, these test suites will be considered in future research to refine this approach.

Table 1. Ontologies for Evaluation

Ontology	Triples	Expressivity	Cls./Prop. /Indv.	Selected rules
Teams[10]	262	$\mathcal{ALCIN}$	9/3/3	16
FOAF[11]	808	$\mathcal{ALCHIF(D)}$	23/69/0	22
mad_cow[12]	1012	$\mathcal{ALCHOIN(D)}$	54/17/13	24
Pizza[13]	3201	$\mathcal{ALCF(D)}$	87/30/0	24
Wine[14]	5710	$\mathcal{SHOIN(D)}$	138/20/206	33

5.3. Data and Analysis

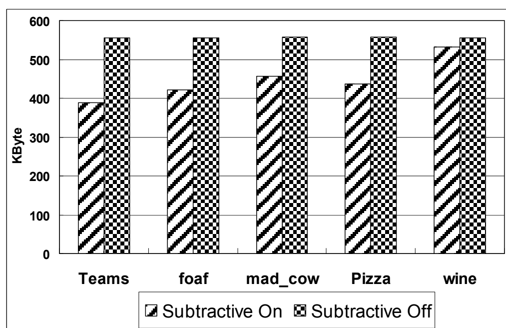

Figure 5. Memory consumption for RETE network building in subtractive mode

The Figure 5 shows the memory consumption for *RETE network building* in subtractive mode. The memory consumptions for all the five ontologies are almost the same (550K) when composability is off and they reduce to some extent after the composability turns on (reduced by 30% for Teams, 24% for foaf, 17% for mad_cow, 21% for Pizza and 4.3% for wine; percentage errors are less than 0.1%). As in subtractive mode this process is only to load a pre-built RETE network and no other actions needs to be performed, the memory consumption is mainly used for store the RETE network and thus memory cost reduction are caused by the reduce of the size of RETE network.

The memory saving shown in Figure 5 is not quite consistent for different ontologies but shows better improvements for ontologies with lower-expressivity. This is mainly because memory saving in this test comes from the removal of unused rules; the reasoning of ontologies of high expressivity requires more rules, which then means less un-used rules and thus less memory saving. For some extreme cases there could be no memory saving as all candidate rules are selected. However, the low processing power of sensors determines that ontology expressivity will in many cases be quite low, which then implies a potential for substantial memory savings with the application of composable reasoner in sensor network.

Memory saving of additive mode is not shown in Figure 5 for better observation as it requires far more memory than subtractive mode. Thus improvement turns out to be not obvious (less than 3%). This is caused by the Drools' high memory cost for on-the-fly rule compilation and RETE network building (more than 10MB), which greatly lowers the impact of memory saved through composability (around 300K).

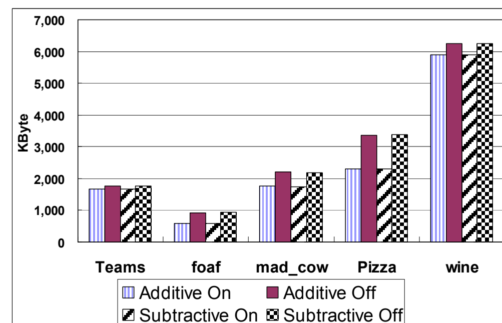

Figure 6. Memory consumption for reasoning

Figure 6 shows the memory cost for the *reasoning process* (i.e. rule evaluation). Note that memory savings for both additive mode and subtractive mode are almost the same for a given ontology. This is because non-select rules are the same for both the two modes, leading to the same RETE network, which determines the memory consumption for reasoning a given ontology. There is a major memory saving for the foaf (reduced by 37%), the mad_cow (reduced by 20%) and the Pizza ontology (reduced by 32%) for both the two loading modes. The memory saving for the teams ontology and the wine ontology, however, are small: circa 5%. Percentage errors for all above listed data are within 1%. There are a number of factors can affect the memory consumption for reasoning, e.g. the size of ontology, the size of selected rule set, the amount of implicit knowledge, etc. Thus it

---

10 http://owl.man.ac.uk/2005/sssw/teams

11 http://xmlns.com/foaf/0.1/

12 http://www.cs.man.ac.uk/~horrocks/OWL/Ontologies/mad_cows.owl

13 http://www.co-ode.org/ontologies/pizza/pizza_20041007.owl

14 http://www.w3.org/2001/sw/WebOnt/guide-src/wine

is hard to draw a clear tendency of memory saving in terms of above listed factors.

Reduction of Execution time for RETE network building after composability is on (as illustrated in Figure 7) is small (within 15%) for additive mode and no obvious or even slightly higher (e.g. for Teams, foaf and wine) for subtractive mode. The improvement for additive mode is because composability reduces the number of loaded rules, which then saves the time spent on rule compilation and RETE network building. The slight increase for subtractive mode is because reasoner under this mode loads a pre-compiled RETE network with a complete candidate rule set no matter composability is on or off, but removes un-used rules only if composability is on, where more time is consumed. Percentage errors for data in this test are within 7%.

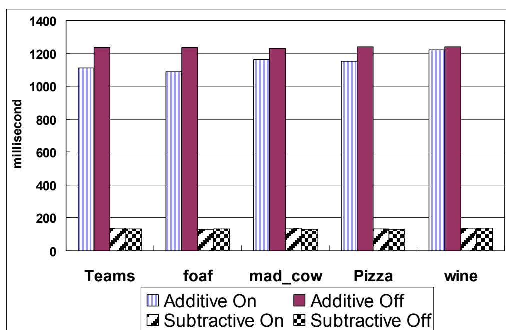

Figure 7. Execution time for RETE network building

As shown in Figure 8, there is no significant reduction on execution time for both the two modes. The Pizza ontology and the team ontology have the most reduction of execution time for additive mode (by 12.4%) and subtractive mode (by 13%) respectively. Except for the Pizza ontology (reduced by 10.7% in subtractive mode), the execution times for other ontologies, although reduced, are all in 10%. Percentage errors for those data are within 16%.

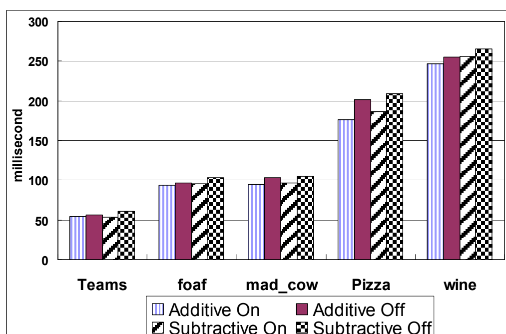

Figure 8. Execution time for reasoning

As shown in Figure 9, for all the five ontologies tested, the memory consumption of the composable reasoner can compete with Jena Micro, which is the smallest Jena reasoner, and is less than the other reasoners. However they use different expressivities, e.g. complete OWL-DL for Pellet, a reduced version of OWL-DL for Jena, and a variation of pD* for the composable reasoner. The comparison of total reasoning time is not listed here for space limit. However generally it (in both modes) is slightly worse than Pellet and Jena. This is indicative of the prototype nature of this implementation and the heavyweight nature of the Drools rule engine.

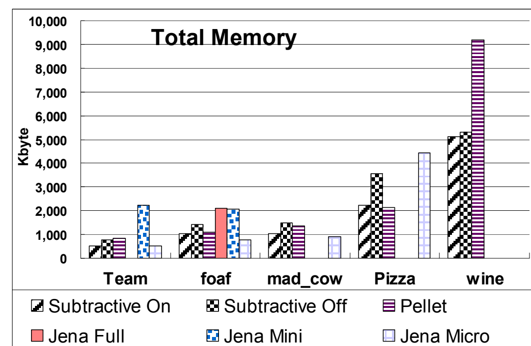

Figure 9. Comparison of total memory

To sum up, the application of automatic composability significantly reduces the memory cost (an average of 27%). However, as yet there is not a significant reduction in terms of time performance. This is mainly because the Drools engine is a heavy weight engine designed as a general rule engine without dedicated optimization for OWL reasoning. Secondly, the current prototypical implementation is lack of optimizations. More work need to be done to address those problems. Although some inefficiencies are found the evaluation result still shows promise in pushing intelligence into local sensor network management. We are currently building the composable reasoner on a light weight rule engine to reduce the time and memory consumption caused by the rule engine itself.

6. Conclusion and Future Work

The objective to push semantics towards the edge of the sensor network is hampered by the lack of availability of lightweight ontology processing and reasoning technologies that are cognisant of the limited resources available in sensor network nodes. This paper proposes an approach to dynamically and automatically compose an OWL ontology reasoner to provide only the level of reasoning required for the ontology in use. This is primarily driven by the

semantic sensor network management scenario described in the introduction section.

A design and prototype implementation for this approach is presented. The automatic composition of the semantic reasoner is facilitated by a modular set of entailment rules, where only appropriate rules are selected. Although implemented using a heavy weight rule engine, our evaluation of this prototype still shows that this approach greatly saves memory (on an average of 27% for subtractive mode). Thus although the time saving is not yet obvious, this approach still seems promising. In addition, the automatic composition feature obviates the need for expert-level input to customise the reasoner for specific deployments.

This prototype does not yet provide support for reasoning on datatype and number restriction; and the lack of consistency checking rules makes this prototype unable to detect inconsistencies in an ontology. These drawbacks will be addressed by future work. It is also planned to extend the rule set to add support for more expressive OWL reasoning, in a modular and composable manner.

As mentioned, this prototype is built on a heavy weight general-purpose rule engine (Drools), where complicate data structures are used for both rules and facts maintenance. This puts some drawbacks on our work such as high memory consumption for runtime RETE network building, etc. Ongoing work is focused on the integration of a lightweight J2ME compatible forward chaining production rule system.

Further research is also required to continue our investigation of the composability of other OWL reasoner technologies, e.g. DL reasoners. It is hoped that further improvements in resource optimisation and reasoning expressivity.

Acknowledgments: This work is supported by the Irish Government under the “Network Embedded Systems” project (NEMBES) as part of the Higher Education Authority's Programme for Research in Third Level Institutions (PRTLI) cycle 4.

References

[1] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli, "A survey of autonomic communications", *ACM Transactions on Autonomous and Adaptive Systems,* 2006.

[2] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, The description logic handbook, Cambridge University Press New York, NY, USA, 2007.

[3] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical OWL-DL reasoner," Web Semantics: Science, Services and Agents on the World Wide Web, 2007.

[4] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, "Jena: implementing the semantic web recommendations", International World Wide Web conference on Alternate track papers & posters, 2004.

[5] D. Tsarkov, A. Riazanov, S. Bechhofer, and I. Horrocks, "Using Vampire to Reason with OWL", International Semantic Web Conference, 2004.

[6] U. Hustadt, B. Motik, and U. Sattler, "Reducing SHIQ-Description Logic to Disjunctive Datalog Programs", International Conference on Principles of Knowledge Representation and Reasoning, 2004.

[7] Y. Zou, T. Finin, and H. Chen, "F-OWL: an Inference Engine for Semantic Web", IEEE Workshop on Formal Approaches to Agent-Based Systems, vol. 3228, 2004.

[8] Z. Wu, G. Eadon, S. Das, E. I. Chong, V. Kolovski, M. Annamalai, and J. Srinivasan, "Implementing an Inference Engine for RDFS/OWL Constructs and User-Defined Rules in Oracle", International Conference on Data Engineering, 2008.

[9] G. Meditskos and N. Bassiliades, "A Rule-Based Object-Oriented OWL Reasoner" IEEE Transactions on Knowledge and Data Engineering, 2008.

[10] E. Amir and S. McIlraith, "Partition-based logical reasoning for first-order and propositional theories," Artificial Intelligence, vol. 162, pp. 49-88, 2005.

[11] B. C. Grau, B. Parsia, E. Sirin, and A. Kalyanpur, "Automatic partitioning of owl ontologies using e-connections," Description Logics, vol. 4, 2005.

[12] H. J. ter Horst, "Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary", Web Semantics: Science, Services and Agents on the World Wide Web, 2005.

[13] Y. Guo and J. Heflin, "A scalable approach for partitioning owl knowledge bases," in Second International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2006), 2006.

[14] O. Kutz, C. Lutz, F. Wolter, and M. Zakharyaschev, "E-connections of abstract description systems," Artificial Intelligence, vol. 156, pp. 1-74, 2004.

[15] P. Hayes, "RDF Semantics", W3C Recommendation, vol. 10, 2004.

[16] J. J. Carroll and J. De Roo, "OWL web ontology language test cases", W3C Recommendation, vol. 10, 2004.

[17] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems", Web Semantics: Science, Services and Agents on the World Wide Web, 2005.

[18] V, Royer and J. J. Quantz, "Deriving Inference Rules for Description Logics: a Rewriting Approach into Sequent Calculi," Technische Universitaet Berlin 1993.

[19] A. Fokoue, A. Kershenbaum, L. Ma, E. Schonberg, and K. Srinivas, "The summary abox: Cutting ontologies down to size," in Proc. of the 5th International Semantic Web Conference, 2006, p. 343.