COROR: A COmposable Rule-entailment Owl Reasoner for Resource-Constrained Devices

Wei Tai, John Keeney, Declan O'Sullivan

Knowledge and Data Engineering Group,
School of Computer Science & Statistics, Trinity College Dublin, Ireland.
{TaiW, John.Keeney, Declan.OSullivan}@cs.tcd.ie

Abstract. OWL (Web Ontology Language) reasoning has been extensively studied since its standardization by W3C. While the prevailing research in the OWL reasoning community has targeted faster, larger scale and more expressive OWL reasoners, only a small body of research is focused on OWL reasoning for resource-constrained devices such as mobile phones or sensors. However the ever-increasing application of semantic web technologies in pervasive computing, and the desire to push intelligence towards the edge of the network, emphasizes the need for resource-constrained reasoning. This paper presents COROR a COmposable Rule-entailment Owl Reasoner for resource-constrained devices. What distinguishes this work from related work is the use of two novel reasoner *composition* algorithms that dynamically dimension a rule-based reasoner at runtime according to the features of the *particular* semantic application. This reasoner is implemented and evaluated on a resource-constrained sensor platform. Experiments show that the composition algorithms outperform the original non-composable reasoner while retaining the same level of reasoning capability.

Keywords: Composable Reasoner, Resource-Constrained Reasoning, OWL Reasoning, Rule-engine Optimization, OWL.

1 Introduction

Quite a few OWL reasoners, using different reasoning technologies, have been developed to provide OWL reasoning services for different purposes. For example some Description Logic (DL) tableau-based reasoners, e.g. Pellet [12], RacerPro [14] and FaCT++ [13], aim to provide sound and complete OWL reasoning services. Some reasoners, e.g. KAON2 [16] and QuOnto [17], are designed to support efficient query services over large data sets. Reasoners such as CEL [18] are specifically dimensioned to provide an efficient subsumption algorithm for some applications (e.g. medical or bio-informatics). Yet more reasoners such as OWLIM [19] and Oracle 11g [22] provide certain levels of embedded OWL (entailment) reasoning services in (large) data stores.

Much of the existing OWL reasoning research aims to develop faster, larger-scale and more expressive OWL reasoners, while there exists only limited work on OWL

reasoners for resource-constrained devices such as embedded devices, mobile phones or sensor platforms. However, as more intelligent embedded systems become pervasive, and with the proliferation of smarter mobile devices, the need for "on-device" semantic reasoning becomes more pronounced, for example, information filtering in context-aware mobile personal information system [28], localized fault diagnoses in wireless sensor networks [29] and context-addressable messaging services in mobile ad-hoc networks [30].

This paper presents COROR, a COmposable Rule-entailment Owl Reasoner for resource-constrained devices. The key contribution of this work is the use of two composition algorithms, i.e. a selective rule loading algorithm and a two-phase RETE algorithm. Instead of selecting a static reasoner configuration, or selecting a-priori from a set of known reasoners or reasoner configurations, composition algorithms try to dimension the OWL entailment rule set and the reasoning algorithm on-the-fly during execution by considering the particular semantic features of the ontology to be reasoned. Our reasoner COROR is implemented and evaluated on a resource-constrained sensor platform (SunSPOT). Experiments show our composition algorithms result in a large reduction in memory and reasoning time while retaining the same amount of reasoning capabilities, freeing up resources on resource-constrained devices or allowing larger ontologies to be reasoned.

This work is currently based on OWL (rather than OWL2) since a de-facto standard OWL2RL rule-set had yet become available. However, the composition algorithms presented in this work are independent of any particular OWL semantic level and they can be equally applied to OWL2 without any fundamental modifications. This selection of a candidate OWL2RL rule-set is subject of ongoing work.

Section 2 presents background and related work. Section 3 details the two composition algorithms implemented in COROR. Details of the implementation are given in section 4, while experiment design, setup and results are discussed in detail in section 5. Section 6 concludes with a discussion of ongoing and further work.

2 Background and Related Work

Background and related work are briefly discussed in this section, including OWL and its sublanguages, the RETE algorithm [5] and some of its optimizations, and finally other resource-constrained semantic reasoners.

2.1 OWL and OWL Sublanguages

OWL is an ontology modelling language standardized by the W3C, consisting of a set of formally defined OWL constructs each of which is given a logic-based semantic [24]. The formal definition of OWL enables reasoning, e.g. entailment computation, to be performed automatically over OWL ontologies. OWL has three standard sublanguages, i.e. OWL-Full, OWL-DL and OWL-Lite, varying in the set of constructs supported, the semantic expressivity, and the complexity of reasoning tasks. Non-standard OWL sublanguages, such as pD* semantics family [4], DL-Lite

family [23] and DLP [51], are also designed for different usages according to the OWL features supported.

In this research we choose the pD* semantics family due to its provision of a definitive entailment rule set and tractable entailment. Some OWL-DL constructs are missing, such as cardinality constructs (cardinality, minCardinality and maxCardinality), some (in)equality constructs (allDifferent and distinctMembers), Boolean combination constructs (unionOf, complementOf and intersectionOf), and oneOf, but still a substantial subset of OWL-DL constructs is kept. Given the resource-constrained context where this work will be applied, any ontology will be generally less complex than OWL-DL. We feel that the pD* family generally have sufficient expressivity and semantics to model our domain to an acceptable degree. COROR is configured to use the pD*sv entailments that extend the pD* semantics with OWL's *iff* semantics for *owl:someValuesFrom*, but at the cost of possibly intractable entailment. Nevertheless, COROR can be configured to use the pD* semantics by simply altering the rule set in use for better computational complexity.

2.2 RETE and RETE Optimizations

The RETE algorithm is a fast pattern matching algorithm for forward-chaining production systems. It forms the basis for most modern production engines, and is the underlying algorithm for COROR. In general RETE builds a discrimination network, termed RETE network, matching and joining facts in the network. A typical RETE network consists of an alpha network and a beta network. The alpha network performs intra-condition matching for individual condition elements in the left hand side (l.h.s.) of each rule. For each rule, successfully matched facts for each condition element, said to partially match the rule, are stored in alpha memory as intermediate results and are propagated into the beta network. In the beta network inter-condition joins are performed by pairwise checking the consistency of variable bindings for intermediate values (i.e. pairwise joins of condition elements). New intermediate results are generated for consistent pairs and they are passed down the beta network for further matching. The final join results that eventually satisfy all condition elements are termed the instantiations of a rule and are added into a conflict set for firing (i.e. fire the r.h.s. action of the corresponding rule). Firing rules may add/remove facts into/from the fact base triggering another RETE cycle as described above.. Multiple RETE cycles are usually required for full entailment of a fact base. The RETE algorithm completes when no more new facts are generated.

Caching intermediate results can substantially speeds up join operations. However an inappropriately ordered sequence of joins can cause very excessive unnecessary memory usage and processing time in the beta network, in particular when two condition elements have no common variables, which leads to a production join. Several heuristics, such as 'most specific condition first', 'pre-evaluation of join connectivity', etc, have been developed to cope with the excessive memory overhead from inappropriate join sequences. Direct application of these can result in optimized RETE networks, however, they have several shortcomings. Firstly direct application of heuristics relies largely on human tuning of the original rule set (e.g. manual order of condition elements in rules) according to heuristics. This is a very onerous task

where reasoners are deployed in an environment with diverse or changing rule sets or changing dataset characteristics. For example in sensor networks different sensors may have different rule sets, and rules may change over time. Secondly, direct application of heuristics [2, 3] usually only considers the rule-set therefore they usually cannot produce optimal join structures for different fact bases [1], where diversity in fact bases and their structure is commonplace in sensor networks.

Researchers have proposed some approaches to automatically optimize join sequences while taking account of the fact base. These optimizations are usually hard to implement and in most cases require modification of the RETE algorithm. However they do not require input from humans and can give different optimizations for different fact bases. Ishida in [1] proposes to use a trial execution before the real execution to collect statistics about the fact set. A predefined cost model is used to evaluate a set of candidate RETE structures and the RETE structure with the minimal cost is selected. This approach can find an optimal RETE structure however its obvious drawback is that a trial execution may not always be practical, particularly where memory, processing ability and power are limited.

Other join structures are also studied to reduce the resource required by RETE network. Work in [10] studies the combination of RETE and TREAT [11] such that the size of beta network can change automatically. The Gator network [50] is proposed as a generalized RETE join network. However Gator and TREAT are not considered as at this stage for COROR as it designed as an experimental reasoner for investigating composition algorithms on rule-entailment OWL reasoners, where the adoption of RETE in rule-entailment OWL reasoners is prevalent.

2.3 Mobile Reasoners

Other work has been devoted to porting semantic reasoning capability onto resource-constrained devices. *MiRE4OWL* [25] is a resource-constrained rule-entailment OWL reasoner developed using C++ for PPC. It adopts two mechanisms to reduce the memory usage of the RETE engine. One is to restrict the number of facts of the same type and the other is to use a primary key to detect duplication of facts and to use an update key to specify the operation to take for duplications. These mechanisms are useful for keeping a light-weight and up-to-date fact base with continuously incoming facts. However its RETE implementation is not optimized and therefore it is likely that inefficient production joins may occur if rules are not tuned by rule experts.

μOR [26] is a resolution-based OWL-DL reasoner for ambient intelligent devices (J2ME CDC compliance). A dynamic rule generation mechanism (similar to the one used in [7]) is used to automatically generate specific inference rules for all concepts/properties/individuals. This approach can construct small specific rules leading to a small (or no) beta network, and scales well for large ABoxes. The drawback, however, is obvious: the size of rule set will increase rapidly with the increase on the size of the TBox.

Bossam [21] is a forward-chaining OWL reasoner for the J2ME CDC platform. However rather than on reducing the runtime memory footprint, Bossam concentrates on providing web-friendly and distributed reasoning.

The above reasoners are the most relevant research to COROR, however their target platform is much less constrained than that of COROR, i.e. SunSPOT (CLDC 1.1 conformant). Some other less related work exists. They are mostly mobile DL tableaux reasoners. *Pocket KRHyper* [27] is a mobile DL reasoner based on hyper tableaux algorithm. Work in [8] introduces an ontology-based context fusion framework for context-aware computing using a sequential rule matching algorithm. Work in [9] discusses *mTaleaux*, a tableaux algorithm for resource-constrained devices. However, they are not directly comparable to our work and due to space considerations will not be discussed in detail here.

3 Composition Algorithms

This section briefly presents our composition algorithms, i.e. the selective rule loading algorithm and the two-phase RETE algorithm that are implemented in COROR.

3.1 Selective Rule Loading Algorithm

The selective rule loading algorithm automatically composes a reasoner rule-base depending on the reasoning capabilities required. It dimensions a selected entailment rule set by estimating which entailment rules are required or desired for reasoning specific ontologies and then selectively loading only these rules into the reasoner. Estimation is performed by comparing OWL constructs used in the ontology against OWL constructs in the l.h.s. of each entailment rule. All OWL constructs used by the ontology are inserted into a *construct set*. Each rule is then individually checked for usefulness. A rule is considered as useful if all OWL constructs used in its l.h.s. are included in the construct set, and it is selected as it *could* be fired for reasoning this ontology. OWL constructs used in the right-hand side (r.h.s.) of each selected rule are then inserted into the construct set as its firing could lead to the insertion of these OWL constructs into the ontology. This process iterates over the remaining unselected rules until all useful rules are identified, while the remaining rules are not used, resulting in a resource saving.

Note that not all selected rules will be fired as the existence of a rule's OWL constructs in the target ontology does not necessitate successful instantiation of that rule. However, unselected rules cannot be fired even if they were loaded due to the absence of relevant OWL constructs in the ontology. A prototype desktop-based implementation and an initial evaluation of this algorithm can be found in [6]. Experiment results show a moderate amount of memory usage reduction but scarcely any reduction in reasoning time in this implementation.

3.2 Two-Phase RETE Algorithm

Rather than optimizing based on reasoning capabilities, as per the selective rule loading algorithm, the two-phase RETE algorithm composes the reasoner at the

RETE algorithm level. A novel interrupted RETE network construction mechanism is adopted that performs only the first RETE cycle immediately after the construction the alpha network (first phase). This enables some information about the ontology to be collected without requiring a full pre-match or traversal of the fact-base. The construction of the beta network resumes after the first-phase matching and a customized RETE network can be composed for the second-phase, tuned for the particular ontology by taking collected information into account. The first RETE cycle resumes after the construction of entire RETE network and the following cycles are performed as in the normal RETE algorithm. The following subsections discuss each phase in detail.

First Phase. In the first phase a shared alpha network is built and the first RETE cycle starts by matching triples against individual rule condition elements in the alpha network. Matched triples are cached in alpha memory awaiting further propagation into the beta network (which is not yet constructed at this stage). A variety of informative statistics about the ontology, e.g. the size of each alpha memory node, the join selectivity factor, etc., can be collected during or after this phase without introducing extra efforts such as specific traversal of the ontology or pre-matching all rules. In our prototype only the number of matched triples for each condition element is gathered. This helps to order beta network join sequences later. An alpha node sharing mechanism is also used to allow condition elements common across different rules to share the same alpha node, thereby reducing the size of the alpha memory and also fact matching time to 1/n for an alpha node (condition element) shared by n rules.

As multiple RETE cycles may be required for reasoning a fact base, information collected at this stage can only be used to optimize the first RETE cycle. However, we notice that for most ontologies that we experimented on (see section 5), the majority of (alpha network) matches and (beta network) joins occur in the first RETE iteration: 15 of a total of 19 ontologies have an average of 75% joins performed in the first iteration (for the remaining 4 ontologies this is still above 50%). Furthermore an average of 83% inferred facts are generated in this iteration. Hence it is appropriate to optimize the RETE network by applying first-cycle optimization heuristics.

Second Phase. In the second phase a beta network is constructed heuristically and the first RETE cycle resumes propagating partially matched intermediate results down through the beta network as condition elements are pairwise joined. However information collected in the first phase enables the application of heuristics to rely not only on characteristics of rules but also on characteristics of the ontology such that a customized beta network (rather than a generally optimized one) can be composed for the particular ontology. Two join sequence optimization heuristics, i.e. the *most specific condition first* heuristic and the *connectivity* heuristic, are implemented in the beta network construction. Their applications are discussed in detail in the following paragraphs.

The ***most specific condition first*** heuristic orders join sequences according to their specificity to avoid long chain effects [3], i.e. where the absence of successful joins is only detected after a large amount of expensive join operations have been performed, leading to a waste of computational resources, in particularly beta network memory. In a previous study [2] Özacar et al assert that using the number of matched facts of a

condition element as a criterion to estimate its specificity can guarantee to find the most specific condition elements. However, this metric can only be calculated after matching, which makes it useless in normal RETE implementations. However we argue that this information can be collected in the first phase of our novel interrupted RETE construction mechanism without introducing extra effort and thereafter it *can* be used here as a straight forward criterion to estimate the specificity: the more facts matched for a condition element's alpha-network node the less specific that condition element is for the particular ontology. A corollary presents where fewer triples match, a condition is more specific. Although the following RETE cycles may affect this specificity ordering (i.e. number of matching facts), it is still sufficient as most joins and intermediate values are generated in the first RETE cycle.

More sophisticated criteria can be introduced for specificity estimation, for example including the cardinality of values to be joined. At this stage we did not implement such heuristics, but the approach taken is equally applicable and, as described later, the approach taken substantially reduces memory and reasoning time.

The ***connectivity*** heuristic ensures that all joining condition elements have variables in common. This prevents product joins and thus can further reduce the amount of intermediate results in the beta network. The connectivity test is performed after the 'most specific condition first' join ordering heuristic: if a condition C_{uncon} is found to be not connectable to all previous conditions C_{pre}, then C_{uncon} is swapped with the first condition after C_{uncon} in the join sequence that *is* connectable to C_{pre}, say C_{con}. As the join sequence has already been ordered by the most specific condition first heuristic, C_{con} is then the one that connects with C_{pre} and with the least specificity of all later connectable conditions. This ensures connectivity in the join sequence and also maintains the specificity ordering of joins where possible.

3.3 Analytical Comparisons between Composition Algorithms

In this section analytical comparisons between the two composition algorithms are presented from three aspects, including reasoning algorithm independence, semantic independence and flexibility in handling changes. Empirical analyses and comparisons can be found in section 5.

As the selective rule loading algorithm constructs a selective ruleset only by analyzing the constructs used in the entailment rules themselves, it is completely independent of reasoning algorithm or ruleset used. This feature enables it to be applied in all forward-chaining rule-entailment reasoners regardless the reasoning algorithms such as RETE or DBMS. Furthermore its application does not need to change the reasoning algorithm and hence it is relative easy to be applied. The two-phase RETE algorithm can only be applied for the RETE algorithm and therefore is not reasoning algorithm independent. In addition its implementation involves changes to the reasoning algorithm i.e. interrupting RETE construction, so is harder to implement compared to the selective rule loading algorithm. Both algorithms are semantic independent. Both the two-phase RETE algorithm and the selective rule loading algorithm are completely independent of the particular ontology or entailment rule set in use.

Addition and deletion are discussed separately with respect to flexibility of handling dynamic changes in the fact-base. Additions can be handled incrementally by the two-phase RETE algorithm due to the intrinsic capability of RETE to handle addition incrementally. Simple deleting facts may lead to logical errors (e.g. deleting facts with different justifications) and therefore may require truth maintenance mechanisms. As truth maintenance is not yet implemented on COROR re-reasoning the entire ontology is required for every deletion. However given that many existing semantic applications only need to reason on static ontology rather than changing KBs, COROR is still sufficient for them. Additions may introduce previously unseen OWL constructs which may cause a problem with the selective rule loading algorithm. In this case re-execution of the selective rule loading algorithm and re-reasoning of the entire ontology are required.

The semantic independence feature of both composition algorithms enables the extension of COROR to support OWL 2 RL without *any* fundamental modification. Given growing adoption of OWL 2 the extension of COROR to support OWL 2 RL is considered as an important task in future work.

4 Implementation

COROR is implemented on the SunSPOT [48] sensor board emulator with SDK v4.0 (blue). The *μJena* framework [49], a cut down J2ME version of *Jena* [15], is used to read and handle OWL ontologies. It provides a powerful interface for ontology access and modification, e.g. parse ontology definitions, support to assert/retrieve OWL axioms, query resources and so on. Rule handling and reasoning are not supported by *μJena*. Given the close connection between *μJena* and *Jena* we ported the *Jena* RETE engine (and relevant modules) into *μJena* rather than implementing them from scratch. As the SunSPOT is only conformant with CLDC 1.1, a subset of J2ME, substantial modifications were required to port the *Jena* rule engine onto *μJena* and SunSPOT.

The selective rule loading algorithm is implemented as a Java class (*RuleSetComposer*) outside of the RETE engine. To enable faster OWL construct identification, instead of analyzing entailment rules at runtime using *μJena* APIs, we manually analyze them beforehand: OWL constructs from both l.h.s. and r.h.s. are identified and coded as rule-construct mappings in a text file which will be loaded and analyzed by the selective rule loading algorithm. A drawback of this approach is it requires different rule-construct mappings to be created manually for different entailment rule sets. The checking for OWL constructs in any ontology is performed automatically at load-time by enumerating the OWL constructs using the *μJena* ontology manipulation API.

The two-phase RETE algorithm is implemented inside the RETE engine (*RETEEngine* class). One problem encountered in the implementation is *Jena*'s extensive use of Java arrays as variable binding vectors where bound values are stored in the corresponding positions in the array as in the rule. This hampers the sharing of common conditions between rules, thereby requiring extensive code refactoring. Four composition modes, i.e. *NonComposable* mode, *Selective Rule Loading* mode, *Two-*

Phase RETE mode, and *Hybrid* mode, are implemented corresponding to the use of no, one or both composition algorithms. In the Hybrid mode both composition algorithms are used and the selective rule loading algorithm first dynamically constructs a selective entailment ruleset for use in the two-phase RETE algorithm.

Entailment is the key reasoning task implemented by COROR. However, some common reasoning tasks can be realized by querying the ontology with all entailments calculated, coined entailment closure (at the moment COROR supports only single-triple-based query). For example, subsumption between two classes *C* and *D* can be reduced to querying the entailment closure with the triple *C rdfs:subClassOf D*, instantiation of *C* as querying with the triple *?x rdf:type C*, where *?x* is a variable and satisfiability of a class *C* as querying with the triple *C rdfs:subClassOf owl:Nothing*, checking if *x* is an instance of class *C* as querying for the triple *x rdf:type C,* and so on. Some other reasoning tasks are not directly supported by this approach. For example, realization of an instance *a* requires finding the most specialized class that *a* instantiates, which requires pairwise subsumption checking for all classes retrieved using *a rdf:type ?x*.

A configuration file is used where users can specify the composition mode, the ruleset to be used, and specify the ontology to be reasoned. Rules are encoded in the *Jena* rule format in a separate rule file, giving users flexibility to modify the rule set, in particular providing simple support for application-specific reasoning.

5 Experiments and Discussions

This section presents and discusses two experiments carried out to evaluate COROR from both the performance and the correctness perspectives.

5.1 Design and Execution

Experiments were performed to investigate the performance impacts of composition algorithms on rule-entailment reasoners.

The memory usage and execution time required to fully calculate entailments of a selected set of ontologies on the SunSPOT emulator (v 4.5.0) is compared for different composition algorithms. These metrics were selected since they directly represent changes in reasoning performance. Some other metrics used to evaluation other OWL reasoners were not selected here as they are not quite suitable for COROR. For example, conjunctive query answering time is not yet implemented in the current version of COROR; reasoning speed on ever enlarging KB is also omitted here as COROR performs load-time reasoning for resource-constrained devices and therefore small or medium ontologies with static sizes are the target of this work. The separate evaluation of individual reasoning tasks such as classification are also not performed in this work as entailment is the key reasoning task in COROR and all other tasks are reduced to querying the entailment closure (as discussed in Section 4).

The memory usage and execution time of COROR (configured to use the hybrid mode) were also compared with other OWL reasoners. As *MiRE4OWL* and *μOR* are not accessible, *Bossam* and three other desktop rule-entailment reasoners, i.e. Jena

2.6.3, BaseVISor 1.5.0 [20], and swiftOWLIM v3.0.1, were selected in this comparison due to their similarity with COROR in terms of expressivity and reasoning algorithm. Note that although Bossam supports J2ME CDC we failed to port it onto SunSPOT as java.util.List is widely used in Bossam while not supported by CLDC 1.1. Jena was configured to used RETE engine only and also the pD*sv rule set. Pellet was also included in this comparison giving readers an intuition of the performance of COROR comparing to a full fledged DL tableau reasoner. As it has proved time prohibitive to port these reasoners onto SunSPOT platform this second evaluation step was performed using a J2SE platform on a desktop computer with Dual Core CPU @ 2.4GHz, Java SE 6 Update 14, and maximum heap size as 128MB (the SunSPOT emulator ran on the same desktop machine).

In total 17 ontologies varying in size and expressiveness were used in our experiments, including: teams [31], owls-profile [32], koala[33], university [34], beer [35], mindswapper [36], foaf [37], mad_cows [38], biopax [39], food [40], miniTambis [41], atk-portal [42], wine [43], amino-acid [44], pizza [45], tambis-full [46] and nato [47]. These ontologies are of small or medium size and are from different domains therefore their usage can avoid any unintentional bias where some OWL constructs are over- or under-used by some ontology designers in different application domains. They are well known and commonly used, and so are relatively free from errors. Due to the low memory and processing power available on SunSPOT only 11 of the 17 ontologies were used in the experiment on the SunSPOT platform while all 17 ontologies were used in the comparison with other reasoners.

5.2 Results and Discussion

The memory usage and reasoning time required by different composition modes on SunSpot COROR implementation are illustrated in figure 1 and 2. Results show that all composition modes use less memory and reasoning time than the NonComposable mode. Note that some tests produced no data e.g. the memory usage required to reason the food ontology in the NonComposable mode, required manual termination before completion due to a long reasoning time (over 30 minutes).

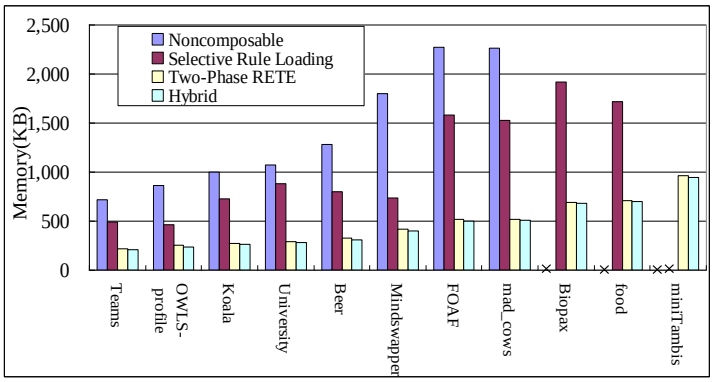

Fig. 1. Memory usage for different composition algorithms (KB).

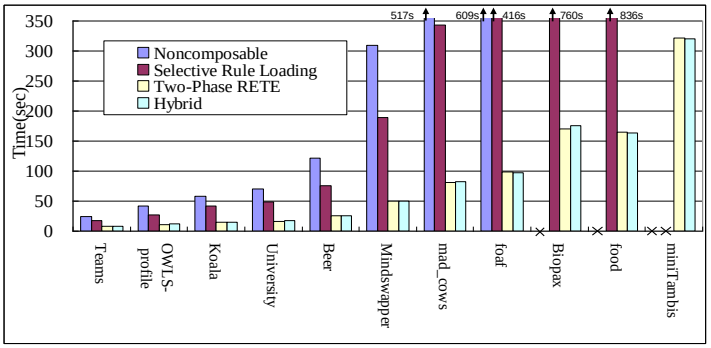

Fig. 2. Reasoning execution time for different composition algorithm (second).

Different composition algorithms vary in their performances. The two-phase RETE algorithm outperforms the selective rule loading algorithm for all tested ontologies. In addition the time/memory reductions in hybrid mode are also limited (comparing to the two-phase RETE mode). We can conclude the reasons for these performance differences by analysing the algorithms, rulesets and ontologies used. The selective rule loading algorithm reduces memory and time by unloading rules. Loaded rules are not optimized. Heuristics used in the two-phase RETE algorithm, however, apply to all rules, even rules that could have been omitted. The two-phase RETE algorithm reorders join sequences of rules according to the number of matched facts of each condition, so any condition element from an unneeded rule with no matching OWL facts is placed at the start of the join sequence. In this way join sequences of unneeded rules can be reordered such that no join operation is needed, as if they are "unloaded". Despite this, some shared alpha network nodes may still be created for the other condition elements of these unneeded rules reducing the size of alpha network, which is not in the selective rule loading algorithm. Hence the two-phase RETE algorithm can have better performance than the selective rule loading algorithm. This also explains the limited benefit in hybrid mode beyond the two-phase mode.

In depth investigations into composition algorithms are performed to identify the sources for the time/memory reductions. Several metrics are selected. As join and match are respectively the two major operations in alpha and beta network, changes on the *number of matches* (*#M*) and the *number of joins* (*#J*) are used to respectively represent changes on the reasoning time in alpha and beta network. Similarly changes on the *number of intermediate results* (*#IR*) generated by matches/joins ($\#IR_M$/$\#IR_J$) are used to represent changes on memory in the corresponding network. By enumerating the *#M*, *#J* and *#IR* values by rule for each ontology under the selective rule loading mode we find that these metrics all drop to zero for unneeded/unloaded rules leading to the reduction of memory usage and reasoning time. For loaded rules these metrics remains the exact same as for the NonComposable mode. These metrics and a close comparison of the results of all modes show that the optimizations applied do not in any way change the results of the entailment process, so the correctness of the process is not affected.

Insights into the two-phase RETE algorithm show that the alpha node sharing mechanism in alpha network contributes most *memory reduction* in this experiment as

the reduced $\#IR_M$ occupies the majority of the total reduced $\#IR$ (an average of 95% of the intermediate result reductions occur in the alpha network for all tested ontologies). However, it is reductions in both the $\#M$ and $\#J$ values (in both the alpha- and beta-networks) that contribute to the decrease in *reasoning time*, but due to the differences in the processing time required for different per match/join operations it is difficult to conclude which contributes more.

Close investigation of the rule set explain the limited memory reduction resulting from the join-reordering in the beta network. The selected pD*sv ruleset is already manually optimized in terms of condition ordering and therefore nearly no automatic optimization is required for this ruleset. This leads to only small reductions in the $\#IR_J$ (and therefore small contribution to the memory reduction). To show that the two-phase RETE algorithm is able to greatly reduce the reasoning time and memory the pD*sv rule set was re-processed to re-order conditions elements for each rule in a sub-optimal manner (as would be typical for user-authored or application specific rules). Tests performed in figure 1 and 2 were re-executed for the rearranged non-optimized rule set and results show that the NonComposable mode required *substantially* more time and memory than it required before. However the two-phase RETE algorithm required the exact same amounts of time and memory as shown in figures 1 and 2, which shows the ability of two-phase RETE algorithm to greatly reduce reasoning time and memory with sub-optimal ordering of rule conditions. These results show that the approach and heuristics selected for COROR are capable of automatically optimizing rules condition ordering to an extent comparable to that of a rule authoring expert.

The comparisons of reasoning time and memory usage between COROR and other state of the art reasoners are given in Figures 3 and 4. The time-based performance COROR is comparable to Jena and BaseVISor. For some small ontologies it runs slightly faster than BaseVISor. Generally Jena requires a longer time to finish its reasoning mainly due to its complicated design to enable flexible ontology manipulation rather than fast reasoning. However, the reasoning execution time of COROR is substantially worse than OWLIM and Bossam.

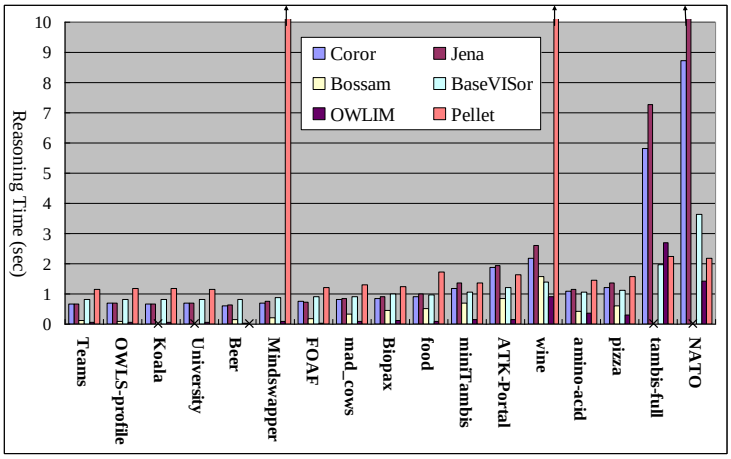

Fig. 3. Comparison of reasoning execution time with other OWL reasoners

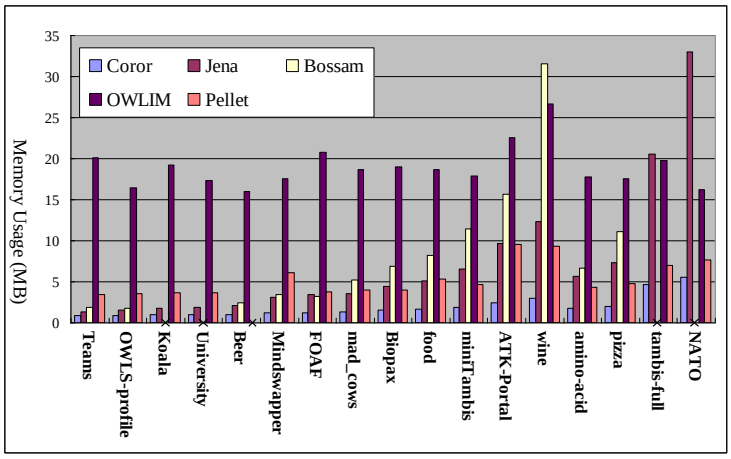

Fig. 4. Comparison of memory usage with other OWL reasoners

On the other hand the memory performance of COROR is much better than the other reasoners. It uses the least memory for all tested ontologies, which indicates much smaller memory footprint can be gained when COROR is applied on resource-constrained devices. Bossam failed for some ontologies so there are no values for them. BaseVISor hides its reasoning process from external inspection so we cannot accurately measure its memory usage and it is omitted from the memory comparison.

The correctness of algorithms (in terms of the pD* semantics) were tested by comparing results of the mode based on the original Jena RETE engine (NonComposable mode) with results from the other composition modes. All the four composition modes generate identical results for all 17 ontologies. Given the tight relationship between the NonComposable mode and Jena RETE engine, we can conclude that our algorithms do not affect correctness.

6 Conclusion and Future Work

We present COROR, a composable reasoner for resource constrained devices. It implements two novel complementary algorithms to compose a custom OWL reasoner at the entailment ruleset level and at the RETE algorithm level. The selective rule loading algorithm establishes a perfect-fit entailment rule subset for the target ontology by selecting only the entailment rules required for that ontology and then loading them into the reasoner. The two-phase RETE algorithm dynamically collects optimization statistics during the reasoning process and uses them to optimize the RETE network building process. This reasoner was implemented on the SunSPOT platform. Experiments show that all combinations of the composition algorithms require less memory and time than the non-optimized version of the reasoner, and require substantially less memory than other off-the-shelf rule-based reasoners.

Further work is actively addressing a number of outstanding topics. First different types of statistics can be collected in the first phase and more sophisticated heuristics

can be designed or selected. For example, as mentioned earlier the heuristics we are currently using to evaluate the specificity of a condition is relatively simplistic, despite its good performance. Secondly the capability to process conjunctive queries needs to be included. This allows richer queries, characteristic of sensor applications. Thirdly, indexing and other join methods, e.g. merge-join, can be studied and tested for better efficiency. Finally, consideration of the recently published W3C OWL2 standard, and the selection of a candidate OWL 2 RL rule set is ongoing.

Acknowledgement

This work is supported by the Irish Government in the “Network Embedded Systems” project (NEMBES), part of the Higher Education Authority's Programme for Research in Third Level Institutions (PRTLI) cycle 4, and by Science Foundation Ireland via grant 08/SRC/I1403 - "Federated, Autonomic Management of End-to-End Communications Services" (FAME).

References

1. T. Ishida, "An optimization algorithm for production systems," IEEE Transactions on Knowledge and Data Engineering, vol. 6, pp. 549-558 (1994)
2. T. Ozacar, O. Ozturk, and M. O. Unalir, "Optimizing a Rete-based Inference Engine using a Hybrid Heuristic and Pyramid based Indexes on Ontological Data," J. of Computers, vol. 2, p. 41 (2007)
3. D. J. Scales, "Efficient Matching Algorithm for the SOAR/OPS5 Production System," Technical Report KSL-86-47, Department of Computer Science, Stanford University (1986)
4. 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, vol. 3, pp. 79-115 (2005)
5. C. Forgy, "Rete: A Fast Algorithm for the many pattern/many object pattern match problem," Artificial Intelligence, vol. 19, pp. 17-37 (1982)
6. W. Tai, R. Brennan, J. Keeney, and D. O'Sullivan, “An Automatically Composable OWL Reasoner for Resource Constrained Devices”, in Proc. Intl. Conf. on Semantic Computing (2009)
7. G. Meditskos and N. Bassiliades, "A Rule-Based Object-Oriented OWL Reasoner," IEEE Transactions on Knowledge and Data Engineering, vol. 20, pp. 397-410 (2008)
8. T. Gu, Z. Kwok, K. K. Koh, and H. K. Pung, "A Mobile Framework Supporting Ontology Processing and Reasoning," in Proc. Workshop on Requirements and Solutions for Pervasive Software Infrastructures (2007)
9. L. Steller and S. Krishnaswamy, "Pervasive Service Discovery: mTableaux Mobile Reasoning," in Proc. Intl. Conf. on Semantic Systems (2008)
10. I. Wright and J. Marshall, "The execution kernel of RC++: RETE*, a faster RETE with TREAT as a special case," Int. J. of Intelligent Games and Simulation, vol. 2, (2003)
11. D. P. Miranker, "TREAT: A better match algorithm for AI production systems," in Proc. of AAAI Conf., pp. 42-47 (1987)
12. Pellet reasoner, http://clarkparsia.com/pellet/
13. FaCT++, http://owl.man.ac.uk/factplusplus/
14. RacerPro, http://www.racer-systems.com/

15.Jena, http://jena.sourceforge.net/
16.KAON2 reasoner, http://kaon2.semanticweb.org/.
17.QuOnto, http://www.dis.uniroma1.it/~quonto/.
18.CEL, http://lat.inf.tu-dresden.de/systems/cel/.
19.OWLIM, http://www.ontotext.com/owlim/.
20.BaseVISor, http://vistology.com/basevisor/basevisor.html.
21.Bossam, http://bossam.wordpress.com/about-bossam/.
22.Oracle Database Semantic Technologies, http://www.oracle.com/technetwork/database/options/semantic-tech/index.html.
23.D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati, "Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family," Journal of Automated Reasoning, vol. 39, pp. 385-429 (2007).
24.P. F. Patel-Schneider, P. Hayes, and I. Horrocks, "Web Ontology Language (OWL) Abstract Syntax and Semantics," W3C Recommendation (2004).
25.T. Kim, I. Park, S. J. Hyun, and D. Lee, "MiRE4OWL: Mobile Rule Engine for OWL," in Proc. Intl. Workshop on Middleware Engineering (ME2010) (2010).
26.S. Ali and S. Kiefer, "µOR - A Micro OWL DL Reasoner for Ambient Intelligent Devices," in Proc. Intl. Conf. on Advances in Grid and Pervasive Computing, (2009).
27.A. Sinner and T. Kleemann, "KRHyper - In Your Pocket," in Proc. Intl. Conf. on Automated Deduction (CADE05), pp. 452-457 (2005).
28.T. Kleemann and A. Sinner, "User Profiles and Matchmaking on Mobile Phones," in Proc. Intl. Conf. on Applications of Declarative Programming for Knowledge Management, pp. 135-147 (2006).
29.R. Brennan, W. Tai, D. O'Sullivan, M. S. Aslam, S. Rea, and D. Pesch, "Open Framework Middleware for Intelligent WSN Topology Adaption in Smart Buildings," In Proc. Intl. Conf. on Ultra Modern Telecommunications & Workshops, (2009).
30.M. Koziuk, J. Domaszewicz, R. Schoeneich, M. Jablonowski, and P. Boetzel, "Mobile Context-Addressable Messaging with DL-Lite Domain Model," in Proc. European Conf. on Smart Sensing and Context (EuroSSC2008) (2008).
31.Teams, http://owl.man.ac.uk/2005/sssw/teams
32.OWLS-profile, http://www.daml.org/services/owl-s/1.1/Profile.owl
33.Koala, http://protege.stanford.edu/plugins/owl/owl-library/koala.owl
34.University, http://www.mindswap.org/ontologies/debugging/university.owl
35.Beer, http://www.purl.org/net/ontology/beer
36.Mindswapper, http://www.mindswap.org/2004/owl/mindswappers
37.FOAF, http://xmlns.com/foaf/0.1/
38.mad_cows, http://www.cs.man.ac.uk/~horrocks/OWL/Ontologies/mad_cows.owl
39.Biopax, http://www.biopax.org/release/biopax-level1.owl
40.food, http://www.w3.org/2001/sw/WebOnt/guide-src/food
41.miniTambis, www.mindswap.org/ontologies/debugging/miniTambis.owl
42.ATK-Portal, http://www.aktors.org/ontology/portal
43.wine, http://www.w3.org/2001/sw/WebOnt/guide-src/wine
44.amino-acid, http://www.co-ode.org/ontologies/amino-acid/2005/10/11/amino-acid.owl
45.pizza, http://www.co-ode.org/ontologies/pizza/pizza_20041007.owl
46.tambis-full, http://www.mindswap.org/ontologies/tambis-full.owl
47.NATO, http://www.mindswap.org/ontologies/IEDMv1.0.owl
48.SUN SPOT, http://www.sunspotworld.com/
49.µJena, http://poseidon.elet.polimi.it/ca/?page_id=59
50.E. N. Hanson and M. S. Hasan, "Gator: An Optimized Discrimination Network for Active Database Rule Condition Testing," Tech. Report, CIS Dept, University of Florida (1993).
51.B. N. Grosof, I. Horrocks, R. Volz, and S. Decker, "Description logic programs: combining logic programs with description logic," in Proc. Intl. Conf on World Wide Web (2003).