# Resource-Constrained Reasoning Using a Reasoner Composition Approach

**Editor(s):** Name Surname, University, Country
**Solicited review(s):** Name Surname, University, Country
**Open review(s):** Name Surname, University, Country

Wei Tai [a, *], John Keeney [b], Declan O'Sullivan [a]
[a] *Knowledge and Data Engineering Group, School of Computer Science and Statistics, Trinity College Dublin, Dublin 2, Ireland*
[b] *Network Management Lab, LM Ericsson, Athlone, Ireland*

**Abstract:** The marriage between semantic web and sensor-rich systems can semantically link the data related to the physical world with the existent machined-readable domain knowledge encoded on the web. This has allowed a better understanding of the inherently heterogeneous sensor data by allowing data access and processing based on semantics. Research in different domains has been started seeking a better utilization of data in this manner. Such research often assumes that the semantic data is processed in a centralized server and that reliable network connections are always available, which are not realistic in some critical situations. For such critical situations, a more robust semantic system needs to have some local-autonomy and hence on-device semantic data processing is required. As a key enabling part for this strategy, semantic reasoning needs to be developed for resource-constrained environments. This paper shows how reasoner composition (i.e. to automatically adjust a reasoning approach to preserve only the amount of reasoning needed for the ontology to be reasoned over) can achieve resource-efficient semantic reasoning. Two novel reasoner composition algorithms have been designed and implemented. Evaluation indicates that the reasoner composition algorithms greatly reduce the resources required for OWL reasoning, facilitating greater semantic data processing on sensor devices.

Keywords: Reasoner Composition, Reasoning, OWL, Rule, Mobile, Resource Constrained, Semantic Web

## 1. Introduction

The marriage between semantic web and sensor-rich systems such as mobile devices and wireless sensor network can semantically link the data related to the physical world with the vast amount of machined-readable domain knowledge encoded on the web. This increases the interoperability and the value of the data, and hence allows applications to have a better understanding of the inherently heterogeneous sensor data through data access and processing on the semantic level. Quite a lot of research has been started seeking better utilization of data in domains in this manner, such as hazard monitoring and rescue [15], context-aware computing [27], environmental monitoring [4], field studies [35], internet of things [18], and so on.

A centralized paradigm is often assumed in such research: all the data processing (such as semantic data annotation, semantic data fusion, semantic data aggregation, decision making, and so on) is handled on a powerful centralized machine either as a local gateway [15] or as remote services on the web [4, 35, 27], and hence the data transmission relies on reliable (and fast) network connections. However, such assumptions do not always hold in reality. In some critical situations such as emergency situations, this infrastructure is prone to interference or damage, causing part of the system with only limited network connections to the central server or even totally detached from the rest of the system. In some other situations such as seafloor monitoring or forest fire monitoring, the surrounding environment could be too harsh (e.g. limited power supply and hostile environment) to be suitable for either an onsite powerful centralized server or a fast internet connection to services in the cloud. In other cases, the overhead involved in relaying raw un-processed data can be very high, and the centralized intelligent data processing overhead at the central processing node acts as a bottleneck in the system. Reality often requires a robust system to be able to retain part (or all) of data processing over the networked devices, e.g. sensor data can be aggregated on devices to meaningful events to minimize network transmission, or decisions/actuations are made locally on the device itself or on a cluster basis [36]. For such a robust semantic-enabled system, it makes sense to push semantic reasoning towards the edge of the system which are lower-specification devices used only for data gathering, actuation, data

aggregation or routing. In the extreme cases they may be only sensors with very limited resources.

Existing semantic reasoners are too resource-intensive to be directly implemented on such resource-constrained devices [3, 34, 38]. This is particularly the case for rule-entailment reasoners. A study in [10] shows the reasoning can take tens or several hundred KB of memory per triple. Hence reasoning over a small ontology can easily drain the resource on a very small device as those deployed on the edge of the systems (e.g. Sun SPOT has 180MHz CPU, 512KB RAM, and 4MB Flash).

Distinct ontologies often vary in characteristics, such as expressivity, the used language vocabularies, structure and so on. The varieties of such characteristics can cause a difference in the requirement of reasoning. A simple example to show this difference would be that reasoning over a RDFS ontology only needs RDFS rules whereas reasoning over full OWL 2 RL ontology requires the entire OWL 2 RL rule set. This difference shows a potential approach for cutting the resource consumption for semantic reasoning. A reasoning approach could automatically adjust itself based on the characteristics of the ontology to be reasoned over to provide "just enough" reasoning for the particular ontology. In this way unneeded reasoning can be avoided, therefore lowering the need for resources in semantic reasoning. This automatic self-adjustment of reasoning capabilities according to characteristics of ontologies is called *reasoner composition* [2].

In our research, we concentrate on the composition of rule-entailment reasoning. This is for two reasons. First, our motivating scenario of resource-constrained reasoning requires domain-specific reasoning to be supported. Modeling and deploying domain-specific rules is the most straight-forward approach. Second, the loose-coupling of rules shows better composability than the other reasoning approaches [2].

Two novel composition algorithms were devised in the course of our research to compose rule-entailment reasoning from two different perspectives: a selective rule loading algorithm composes on the rule set, and a two-phase RETE algorithm composes in the reasoning algorithm. The composition algorithms were implemented in a proof-of-concept resource-constrained OWL reasoner (COROR). As the goal of this work was to reduce resource consumption for embedded rule-entailment reasoning, our evaluation strategy concentrated on memory and time performance measures. Results indicate that our reasoner composition algorithms significantly reduce the resource consumption required for rule-entailment reasoning, thus facilitating semantic reasoning in resource-constrained environments.

The paper is organized as follows. Section 2 presents related work, including a categorization of existing OWL reasoning approaches and a discussion of the optimizations and composition approaches used by existing resource-constrained reasoners. Section 3 elaborates the design of the two composition algorithms. Implementation details for a prototype implementation ("COROR") are provided in section 4. Section 5 follows with an evaluation of our work. Section 6 concludes with a discussion of the significance, limitations and directions for future work of the research.

## 2. Background and Related Work

This section discusses the background and related work from three aspects. Firstly, a categorization of OWL reasoning approaches is outlined and a rationale is provided for choosing forward-chaining rule-entailment reasoning as a base technology for our work (section 2.1 and 2.2). A second aspect is a discussion of the optimizations or composition approaches (if any) adopted by the existing resource-constrained OWL reasoners (section 2.3). Finally, a discussion of the main mechanism for implementing a rule engine – RETE – is presented, along with some RETE optimizations from which some composition algorithms are inspired (section 2.4).

### *2.1. A Categorization of OWL Reasoning Approaches*

Considering the quite different reasoning approaches adopted in OWL reasoners, a categorization of OWL reasoning approaches is important (1) to provide us with a grounding in OWL reasoning approaches, and (2) to enable the investigation of reasoner composition based on a category of reasoning approaches rather than a particular reasoner implementation. A survey of the reasoning approaches was conducted among a group of 26 OWL reasoners from the state of the art. By examining the reasoning approaches used, a categorization was developed such that the reasoners could be classified into five categories, i.e.

*Description Logic (DL) tableau reasoners*, (*forward-chaining) rule-entailment reasoners*, *resolution-based reasoners*, *reasoners using hybrid approaches,* and finally *reasoners using miscellaneous approaches*.

**DL tableau reasoners** convert OWL entailment reasoning to DL knowledge base satisfiability tasks based on the semantic connections drawn in [21]. Satisfiability checking is performed by a tableau engine through constructing a model for the DL knowledge base using a set of consistency preserving *transformation rules*. If a model is found, then the DL knowledge base is said to be *satisfiable*, and *unsatisfiable* otherwise. In this sense, for example, to check that if an OWL inclusion axiom

(ex:Car rdfs:subClassOf ex:Vehicle)

is entailed by an ontology $O$ will be converted into checking the satisfiability of a DL concept

$$\neg \mathrm{Car}^{\mathrm{T}} \sqcup \mathrm{Vehicle}^{\mathrm{T}}$$

is satisfiable in $O^{\mathrm{T}}$ ($\mathrm{Car}^{\mathrm{T}}$ and $\mathrm{Vehicle}^{\mathrm{T}}$ are the corresponding DL concepts in the translated knowledge base $O^{\mathrm{T}}$). State of the art reasoners falling into this category include FaCT++ [45], Pellet [53], RacerPro [54], and HermiT [55].

**Rule-entailment reasoners** perform rule-based OWL reasoning using (forward-chaining) production systems. In rule-entailment reasoners OWL reasoning will be performed using a set of production rules called *entailment rules*. Depending on how the ontology will be handled the entailment rules can be either *description logic programs* (DLP) style [19] where OWL semantics are implied in rule constructors, or *pD** style [44] where dedicated non-logical symbols need to be introduced to explicitly represent OWL language constructs. For the DLP style of rules the entire ontology is often translated into a mixed set of (ontology-specific) entailment rules (as DLP rules) and facts. For the pD* style of rules, ontologies are parsed as facts and the reasoner will match the facts against a set of ontology independent entailment rules (e.g. pD* rules). Reasoning is the same for both types: a production rule engine is used to load the (pD* style of DLP style) rules and calculate an entailment closure of the ontology using the (parsed) facts. Most reasoners of this type use the RETE algorithm, (the most popular fast pattern matching algorithm for production rule engines [5]) to perform the match process. Reasoners of this category type include BaseVISor [28], OWLIM [9], Bossam [24], MiRE4OWL [26], O-DEVICE [52] and the Jena forward chaining engine [12].

**Resolution-based reasoners** perform OWL reasoning using resolution engines which could be a Prolog engine [40, 41], a Datalog engine [16] or a full First Order Logic (FOL) theorem prover [25]. Depending on the type of resolution engine in use, a resolution-based reasoner translates the ontology into a program of the corresponding logic, such as FOL [25], Prolog [40, 41], Datalog [16] and so on. Due to the close connections between rules and horn clauses, the rule-based ontology reasoning is also widely adopted by resolution-based reasoners. However, the translated ontology (facts and entailment rules) is reasoned in a backward-chaining manner using a resolution engine rather than in the forward chaining manner as in rule-entailment reasoners. F-OWL [47], Surnia [20], Thea [40] and the Jena backward-chaining engine [12] are resolution-based reasoners performing pD* style rule-based reasoning. Resolution-based reasoners using DLP style rule-based reasoning include KAON2 [56] and Bubo [41]. Hoolet [25] is a resolution-based reasoner but does not perform rule-based reasoning, instead, the OWL ontology is translated into a FOL theory and then reasoned over using a FOL theorem prover.

Some other reasoners are devised to provide efficient reasoning for a particular subset of OWL or for some specific purposes or tasks, and they use dedicated and less general reasoning approaches. These reasoners are classified as **miscellaneous reasoners** in our categorization. For example, CEL [30] implements a polynomial time classification algorithm designed for OWL 2 EL [13], an OWL 2 fragment that is mainly used by health-informatics/bioinformatics ontologies such as SNOMED CT [33] and Gene ontology [31]. ELK [11] is also an efficient OWL 2 EL reasoner that performs classification for OWL 2 EL ontologies by computing concept saturation using a set of inference rules. A series of optimizations are employed to achieve a good reasoning performance, such as indexing and tabling of concepts, concurrent axiom processing, and avoiding repetitive application of decomposition rules. One thing worth noting is that the word “composition” in the composition/decomposition optimization in ELK is interpreted differently from in this work since composition in ELK means to bring together atomic concepts into complex subsumption axioms and the opposite for decomposition. Reasoners such as Owlgres [39] and QuOnto [1] use query rewriting algorithms to efficiently answer queries over large

datasets of OWL 2 QL expressivity [13]. The SPIN approach uses SPARQL CONSTRUCT and SPARQL UPDATE to handle OWL 2 RL reasoning [17].

Some reasoners combine two or more reasoning approaches presented above to exploit the advantages of each approach. These reasoners are categorized as **hybrid reasoners**. Some well-known hybrid reasoners include Minerva that combines the DL tableau approach with the resolution-based approach [48], DLEJena that combines the DL tableau approach with the rule-entailment approach [29], Jena that combines rule-entailment approach with resolution-based approach [12], and Pellet that incorporates a forward-chaining rule engine for rule handling. The work presented in [51] introduces a modular ontology classification DL tableau approach: to reason an ontology falling into a very expressive DL language *L'*, rather than directly using an *L'-reasoner*, which is usually slow, a module whose expressiveness falls into a lesser DL language, say *L*, is extracted, which can then be classified more efficiently (e.g. in polynomial time using an *L*-reasoner). The obtained pre-computed subsumptions are then fed back to the *L'-reasoner* to completely compute all remaining subsumptions for the input ontology.

A similar categorization of OWL reasoning approaches has already been drawn in [47] where OWL reasoning approaches are categorized into three categories according to the different underlying logics. They are (1) a *DL tableau* type that reasons OWL using DL tableau reasoners (e.g. FaCT++, RacerPro, Pellet and HermiT), (2) a *full FOL* type that translates OWL ontologies into FOL and does the inference using full FOL provers (e.g. Hoolet and Surnia), and (3) a *partial FOL* type that translates OWL into a language/logic implementing partial FOL (e.g. Datalog, production rules, Prolog, F-logic) and then uses a (specialized/general) partial FOL engine to perform OWL reasoning (e.g. OWLIM, Jena, F-OWL, KAON2).

A major weakness of the classification in [47] in terms of our research is that the different reasoning algorithms used by different category are not distinguished: the partial FOL type covers both the rule-entailment type and the resolution-based type where the reasoning algorithms are disparate. However it is important that they are distinguished in this research as the different reasoning algorithms may need different composition mechanisms. In addition, new OWL reasoning approaches have emerged in the past few years requiring a new classification.

### *2.2. Rule-Entailment Approaches as the Target of the Reasoner Composition Research*

Among the five reasoning approach categories presented in this paper, rule-entailment type reasoning was chosen as the target for our reasoner composition research. This decision was made by considering both the ease for composition and the suitability of this type of reasoner to operate in resource-constrained environments.

Most rule-entailment reasoners perform reasoning using static pD* style entailment rules (to the best of our knowledge only one rule-entailment reasoner, i.e. O-DEVICE, uses the DLP style rules). Compared to DLP style rules, which are ontology specific, each entailment rule of the pD* style only carries a small piece of the OWL semantics for a particular OWL language construct. Hence, using pD* style rules enables fine-grained entailment rule selection based on the amount of semantics used in the ontology: with this approach only rules implementing the OWL constructs *required* by the target ontology should be applied in the reasoning process, and the other rules can be removed, especially those that can be determined a-priori to never successfully complete for the given target ontology.

In terms of the suitability of rule-entailment approach for resource-constrained environments, firstly, rules are already widely used on devices to perform data processing and device management, the rule-entailment approach have the intrinsic capability to additionally execute non-semantic rules. Secondly, candidate OWL reasoners have already been implemented for resource constrained environments [24, 26]. In addition, compared to backward-chaining used by resolution-based reasoners performing goal-directed reasoning, the forward-chaining feature of the rule-entailment approach allows corresponding actions to be triggered asynchronously based on events. This is also part of the reason for the popularity and success in applying (non-reasoner) forward-chaining production rule engines to small devices [14, 42].

Backward-chaining resolution-based approaches also show good potential for composition based on their rule-based reasoning feature. DLP style resolution rules can be an exact rule translation of the OWL ontology to be reasoned over. The composability of resolution-based reasoners using

pD* style rules can be also composable in a similar manner as the selective loading of pD* style rules in rule-entailment reasoners. However, as discussed earlier networked embedded systems usually require rules to handle events periodically, hence the data-driven characteristic of forward-chaining reasoning of rule-entailment reasoners show better suitability than the goal-directed characteristic of resolution-based approach.

Rules (transformation rules) are also used in tableau-based approaches to decide the satisfiability of the knowledge base. However, as they are transformation rules that describe procedures of DL tableau calculi, so a selective rule application will harm the completeness of the algorithm [38]. Furthermore, compared to the plain text formats of entailment rules, DL transformation rules are often hardcoded, further impeding their dynamic composition. In addition, to relieve the large time/memory complexity of DL tableau calculi, a wide range of complex, code-level optimizations are entwined in the application code to achieve efficient practical DL reasoning, such as different backtracking mechanisms, loop detection mechanisms, and model caching techniques [7]. These optimizations would need to be adjusted at runtime to comply with the composition algorithm, which makes the dynamic composition of DL tableau reasoners very complicated.

Those reasoners described as miscellaneous approaches are designed for efficient reasoning for specific reasoning tasks, specific ontology expressivities, or specific application areas rather than for general-purpose OWL reasoning. Hence their application- or domain-specific crafted and hardwired algorithms make them less appealing as candidates for composability studies. Meanwhile, the composability of a hybrid reasoner relies on each individual component reasoning approach used, where composability for each approach has already been discussed above.

In summary, the rule-entailment approach was considered the most suitable approach for carrying out our reasoner composition research for resource-constrained OWL reasoning.

### *2.3. Resource-constrained OWL Reasoners and Compositions*

The existing OWL reasoning approaches are mainly designed to run on desktop or server computers and hence their resource-intensive features hinder their direct migration onto resource-constrained devices where resources are modest or scarce [34, 3]. Some dedicated resource-constrained OWL reasoners have been developed and many optimizations have been incorporated to reduce especially the memory footprint. This section discusses some of the more notable existing resource-constrained OWL reasoners including MiRE4OWL, Bossam, μOR, mTableaux and pocket KRHyper and their optimizations.

Mire4OWL [26] is designed for context-aware applications and it implements the rule-entailment approach. It proposes to maintain a light weight fact base by keeping only the latest context facts and makes extensive use of indexing to speed up duplication checking [14, 26].

μOR [3] uses a dynamic rule generation mechanism to generate ontology-specific DLP style *explicit rules* (i.e. rules for reasoning over individuals) from predefined OWL rule patterns. It uses a simple pattern matching and resolution algorithm for rules evaluation (not the logic resolutions as used in the resolution-based approach) [3]. This algorithm has been proven to use only a small amount of memory and time to reason over very small ontology (100 – 300 triples). However, there is no evidence showing it will scale well for larger ontologies where the number of dynamically generated rules will grow rapidly, possibly leading to a substantial degradation of the reasoning performance. Furthermore, this approach is not extensible to *implicit rules* (rules for terminological reasoning and expressed as the pD* style) and domain-specific rules. Hence other rule sets are still loaded as a whole without considering the semantics of the ontology.

mTableaux [38] realizes a mobile version of DL tableau calculi and incorporates three optimizations designed specifically for DL tableau calculi to reduce resource consumption. They include *selective consistency rule application*, *disjunction skipping* and *weighted disjunction/terms expansion rule* [38]. The former two optimizations customize the application of transformation rules according to the ontology however with a compromise on the completeness of the reasoner.

Pocket KRHyper [37] is a FOL theorem prover using hyper tableau calculi [8]. A DL interface is built into Pocket KRHyper allowing DL axioms to be reasoned over as FOL formulae. Two optimizations are employed to conserve resources including the use of different DL/FOL

transformation schemes for different parts of the knowledge base (with different variability) and the use of different treatments for interests and disinterests.

Bossam [24] also implements the rule-entailment approach however there is no evidence that Bossam implements optimizations to maintain a small memory footprint for resource-constrained devices.

Among the five resource-constrained reasoners, only μOR and mTableaux consider the ontology to be reasoned over while applying their optimizations whereas the optimizations for the rest, including the two rule-entailment reasoners Bossam and Mire4OWL, are static. Hence there is no known composable resource-constrained rule-entailment reasoner.

### *2.4. An Overview of RETE and Optimizations*

The RETE algorithm [5] is the typical reasoning algorithm used for the rule-entailment approach and forms the underlying reasoning algorithm for most rule engines and is used in most of the rule-entailment reasoners including the Jena forward engine, BaseVISor, Bossam, MiRE4OWL and O-DEVICE. It is designed as a fast pattern matching algorithm for forward-chaining production systems. RETE caches intermediate matching results. Although this mechanism allows efficient rule matching, it can sometimes use excessive amounts of memory when rules are authored in an inappropriate manner. This can cause particular problems for resource-constrained devices. Optimizations have been developed to cut RETE memory footprint, but most optimizations are performed statically by tuning rules. Some optimizations do take the characteristics of fact base into consideration and hence show potential for reasoner composition. This section discusses the RETE algorithm and also investigates the possibility of using existing RETE optimizations for non-reasoner applications as part of a reasoner composition mechanism to achieve efficient resource-constrained reasoning.

The RETE algorithm performs rule matching using discrimination networks (often termed *RETE networks*) constituted by inter-connected one-input/two-input nodes (Refer to figure 8a for an example). A one-input node (often termed *alpha node*) performs *intra*-condition checks of the input facts against a rule condition from the left hand side (l.h.s.) of a rule, and each intra-condition check is termed a *match* operation. Successfully matched facts for each condition part of each rule are cached in the *alpha memory* attached with the corresponding alpha node as *intermediate results* (IRs). All alpha nodes together form the *alpha network* of a RETE network. A two-input node (termed *beta node*) performs pair-wise inter-condition checks for consistent variable bindings between the IRs generated by the next alpha node in the rule and IRs generated from successful joins by the previous beta nodes. Each inter-condition check is termed a *join* operation. A beta node has a *beta memory* consisting of a left beta memory storing IRs received from the previous alpha or beta node and a right beta memory storing IRs from the next condition in the rule. When a join is successful for a pair of input IRs a new output IR will be constructed and passed down to the next beta node in the network. The inter-connected beta nodes form the *beta network* of a RETE network. Output from the last beta node is a successful match of the entire l.h.s. of a rule and is termed an *instantiation* of the rule. A rule is said to be *triggered* once an instantiation is generated, and actions specified in the right hand side (r.h.s.) of the rule are then performed. Actions are included in a *terminal node* attached to the last beta node, and some context information is also stored in the terminal node for triggering the actions. Once the actions are successfully executed, the rule is said to be *fired*. For rule-entailment reasoners, the only action is the construction and insertion of inferred facts into the fact base. Newly inserted inferred facts would possibly re-trigger some rules, causing the iterative execution of RETE until no more rules are triggered and no more facts can be generated.

Caching IRs will lead to a rapid growth in memory consumption when the join sequences are inappropriately arranged causing few shared variables between two joining conditions. An extreme case would be that two condition elements have no common variables, leading to Cartesian production joins and hence a drastic increase in the memory storing IRs. To cope with this problem, optimization heuristics are incorporated to re-order RETE join sequences, two major approaches are the '*most specific condition first'* heuristic [22, 46, 23, 32] and the '*pre-evaluation of join connectivity'* heuristic [50, 22, 23, 32]. The most specific condition first heuristic is derived from the rule of thumb that the more specific a condition in a rule the more likely it is to have fewer matched facts, and

therefore pushing a more specific condition towards the front of a join sequence is more likely to generate less IRs in a beta network [46, 32]. The pre-evaluation of join connectivity heuristic re-orders join sequences to avoid Cartesian product joins as much as possible: it is regarded that a condition should have at least one common variable with previous conditions where possible (i.e. Cartesian product joins, if any, are pushed to the end of the join sequence, where there will be fewer input IRs).

Heuristic criteria have been derived to determine the specificity of a condition. For this research we mainly describe a most relevant work in [32] where three criteria are designed for OWL entailment rules. The criteria are *counting matched facts* for conditions (the more matched facts of a condition the less specific it probably is), *counting variables* for each condition (the more variables a condition has, the less specific it is), and *analyzing the complexity of OWL predicates* (the more complex the OWL predicate in a condition, the more specific the condition is).

Although these criteria have potential to work well, there are some drawbacks with the above criteria when applied, which hinders their direct application to construct better join sequences. Reordering the join sequence according to the number of matched facts for each condition is considered as the most effective in [32] and it complies with the idea of reasoner composition. However it is regarded to be a paradox: the RETE network is not yet constructed when rules are optimized but counting the matched facts for a condition requires the rule to be fully matched. An obvious drawback with the criterion of using the number of variables to determine specificity is the difficulty in determining the specificities when two conditions have the same number of variables. This is especially a common case for OWL entailment rules (e.g. pD* or OWL 2 RL) where conditions are often expressed using triple patterns, i.e. (subject predicate object) so every condition can only have 1, 2 or 3 variables. The third heuristic is specifically designed for rule-entailment OWL reasoning approaches. However it is static and unable to determine the specificity when two conditions share the same predicates. For example, the property *rdf:type* is usually associated with the *owl:Class* construct, which occurs in almost all OWL ontologies, while *rdf:type* can also be associated with *owl:TransitiveProperty*, which may not exist for many ontologies. Furthermore, it does not consider the particular ontology to be reasoned over which may make extensive use of particular constructs.

To complement heuristics to construct better join sequences, some systematic approaches have been developed to combine multiple heuristics [50, 22, 23, 32]. Still many of them are static and consider only rules [50, 32], and as is indicated in [23] such static approaches usually cannot produce optimal join structures for different fact bases. The work in [23] suggested producing better join sequences using a more dynamic approach to apply heuristics by taking the characteristics of the fact base into account. This dynamism is achieved in [23] using full a-priori execution: join costs for every possible join structures need to be estimated in the pre-execution according to a predefined cost model, and the join structure with the least cost is then regarded as the optimal one. Rather than to optimize directly, RETE optimization heuristics are used in this work as constraints to control the enumeration of join structures. Experiments show later that this approach reduces the amount of join operations, leading to fewer IRs, and so less memory usage, whereas a pre-execution may not be realistic in a resource-constrained environment considering the large amount of resource consumption (memory, time and battery) incurred by the pre-execution. Furthermore the evaluation of cost for a large number of enumerated models is too computational-intensive to be performed on small devices. Hence reasoner composition approaches need to be designed for RETE to run on resource-constrained environment.

To clarify, the terms “condition” and “alpha node” can be used interchangeably, however, the term “condition” is mostly used when describing rules and the term “alpha node” is mostly used when describing a RETE network.

## 3. Rule Entailment Reasoner Composition

For the reasons outlined in section 2, the optimizations used in existing resource-constrained devices and optimizations developed to cut RETE memory usage in the state of the art are insufficient to compose rule-entailment reasoners for resource-constrained environments. Hence dedicated composition mechanisms are required.

Given the fine-grained semantics of each OWL entailment rule and the various expressivities of different ontologies, a natural intuition for reasoner

composition would be to selectively apply entailment rules. For example, for a very expressive ontology such as the *wine* ontology (with a SHOIN(D) DL expressivity) all the OWL entailment rules would be required in the reasoning, but a less expressive ontology such as *pizza* (with a ALCF(D) DL expressivity) would not use as many OWL constructs and hence some OWL entailment rules such as the rules for handling transitive properties, number restrictions, sub-properties, are not required. By removing these rules from the reasoner's rule set, fewer resources are needed for reasoning. In particular, although non-required rules will never successfully trigger, they may require extensive memory at the RETE alpha-network construction stage and the early stages of the RETE beta-network join process even though the later parts of the beta-join sequence will never successfully complete. A similar rule selection process is also used for selective consistency rule application by mTableaux.

Another natural intuition for reasoner composition is to include the particular characteristics of the ontology to be reasoned over into the application of RETE optimization heuristics. This is inspired by a related work conducted in [23]. However, as mentioned the use of a-priori execution to collect characteristics of the fact base uses resources which must be avoided for resource-constrained environment. Hence some mechanism is needed to allow ontology characteristics to be gathered without using a-priori execution, e.g. during the constructions of a RETE network.

As a result of the above intuitions, *two* novel reasoner composition mechanisms have been designed: a *selective rule loading* algorithm that selectively applies rules according to the OWL constructs included by the ontology to be reasoned over, and a *two-phase RETE* algorithm that uses an new interrupted version of the RETE algorithm to collect ontology-specific statistics, and then applies appropriate optimizations based on the collected statistics. These two composition algorithms are respectively elaborated in section 3.1 and 3.2.

As some of the features of pD* semantics suggest that it is an appropriate semantics to be implemented in resource-constrained environments, the pD* semantics [44] is used to illustrate the composition mechanisms. First, it implements a subset of the OWL semantics as RDFS-like rules. Although some OWL constructs are missing, it still preserves a substantial subset of OWL-DL constructs, including value restrictions, class axioms, property relations, property cardinality restrictions, logical characteristics of properties, and some individual identity. Given the low resource availability on resource-constrained devices, ontologies deployed on them will generally be much less complex than OWL-DL and hence pD* should provide sufficient expressivity for resource-constrained reasoning. Second, the pD* semantics are designed to have relatively low entailment complexity, i.e. PTIME entailment complexity when variables are not used in the target ontology and NPTIME entailment complexity when variables are used in the target ontology.

However as will be discussed in section 3.3, the composition mechanisms are designed to be independent of semantics and therefore are not limited to pD* semantics. The OWL 2 RL rule set has the similar characteristics as the pD* rule set in terms of low computational complexity and good composability, and hence it also shows good suitability for resource-constrained environments. However as will be discussed in detail in the end of section 4, considering the difficulties involved in implementing an efficient OWL 2 RL rule set and the small contribution to the main aim of this research to demonstrate the composition algorithms can reduce resource consumption of reasoning we opt for pD* to ease our proof-of-concept implementation.

### *3.1. Selective Rule Loading Algorithm*

In general the selective rule loading algorithm dimensions a selected entailment rule set by estimating the usage of each entailment rule using predefined *rule-construct dependencies*, which describe the containment of OWL constructs in rules. To further explain how rule-construct dependencies work some concepts are defined. An OWL construct referred to in the l.h.s. of a rule is termed a *premise construct* of the rule and the rule is said to *depend* on the construct to fire; a construct in the r.h.s of a rule is said to depend on the rule and the construct is termed a *consequence construct* of the rule. For example, for the rule entitled rdfp13c given below in figure 1, rdfs:subPropertyOf is a premise construct of the rule and owl:equivalentProperty is a consequence construct. Some rules may have multiple premise constructs and consequence constructs.

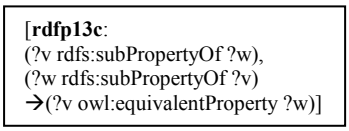



Fig. 1. Rule rdfp13c

Rule-construct dependencies are used in the selective rule loading algorithm to determine if a rule needs to be selected for the reasoning process. It is not difficult to see that a necessary condition for a rule's firing is that all its premise constructs appear in the ontology. Hence a rule is selected only if all its premise constructs are included in the ontology. The successful firing of the rule may introduce new facts, where these facts (containing consequence constructs) may introduce new constructs that were not present in the original ontology, so once a rule is selected, its consequence constructs are added to the premise construct set and rule selection continues. Continuing with the above example, the successful firing of the rule rdfp13c will insert owl:equivalentProperty into the particular ontology's fact-base, which will then cause rule rdfp13a and rdfp13b to be selected (as given in figure 2). This problem can be addressed using repetitive selection.

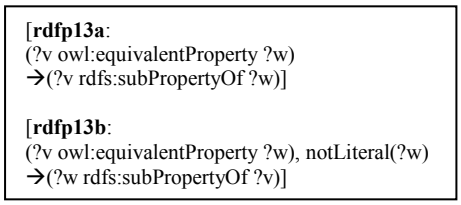



Fig. 2. Rule rdfp13a and rdfp13b

Based on the above discussion, the selective rule loading algorithm is then formalized into three steps.

- First, gather all OWL constructs included in the ontology to be reasoned over and insert them into an *ontology construct set*.
- Second, select those rules from the chosen rule set whose premise constructs are included in the ontology construct set and insert the selected rules into a *selected rule set*.
- Third, insert the consequence constructs of the rules selected in step two into the ontology construct set. Repeat steps two and three until no more new rules are selected.

Rules included in the *selected rule set* when this algorithm terminates is then the set of rules required for the particular ontology to be reasoned over. Rules not selected must have premises out of the scope of the ontology and therefore are not used in the reasoning, thus minimizing resource usage (e.g. memory and CPU time) for both the construction of RETE networks and the fact matching processes, without any loss of accuracy in the reasoning process.

### *3.2. Two-Phase RETE Algorithm*

The *two-phase RETE* algorithm is inspired by the idea of exploiting the characteristics of the fact base during RETE join sequence optimization, as described in [23]. However, instead of introducing extra effort for statistic collection (e.g. a-priori RETE execution) the two-phase RETE algorithm uses a novel *interrupted RETE construction* mechanism that uses experience gained and statistics gathered during the alpha network construction phase (*first phase*) to inform the beta network optimization and construction phase (*second phase*). In this case the optimizations referred to by previous works [23, 32] can be selectively or jointly applied to generate a more efficient RETE network for the *particular* ontology to be reasoned over. To elaborates both phases with an example a simple ontology and two example rules (rdfs9 and rdfp15 from the pD* rule set) are used (figure 3). The ontology defines a *car* concept is a sub-concept of *vehicle* and the car property *hasEngine* is a *someValuesFrom* restriction on the *hasComponent* property. Asserted facts are labeled from T1 to T12, and facts I13 and I14 are inferred according to the rule rdfs9 and rdfp15.

***Rules***
[rdfs9: (*?v* rdfs:subClassOf *?w*), (*?u* rdf:type *?v*) →(*?u* rdf:type *?w*)]

[rdfp15: (*?v* owl:someValuesFrom *?w*), (*?v* owl:onProperty *?p*), (*?u ?p ?x*), (*?x* rdf:type *?w*) →(*?u* rdf:type *?v*)]

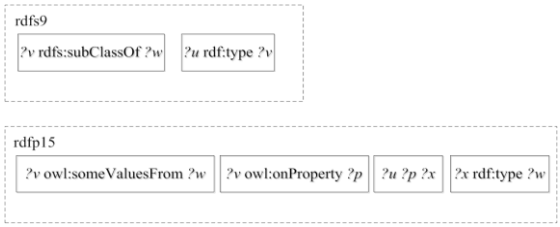
***Ontology***

| | |
|---|---|
| ex:Car rdf:type rdfs:Class. | T1 |
| ex:Car rdfs:subClassOf ex:Vehicle. | T2 |
| ex:Fiat rdfs:subClassOf ex:Car. | T3 |
| ex:Engine rdf:type rdfs:Class. | T4 |
| ex:hasEngine rdf:type owl:Restriction . | T5 |
| ex:hasEngine owl:onProperty ex:hasComp. | T6 |
| ex:hasEngine owl:someValuesFrom ex:Engine. | T7 |
| ex:myCar rdf:type ex:Car. | T8 |
| ex:azrTurbo rdf:type ex:Engine. | T9 |
| ex:myCar ex:hasComp ex:azrTurbo. | T10 |
| ex:myCar ex:hasComp ex:alcon . | T11 |
| ex:myCar ex:hasComp ex:energyMX1. | T12 |
| | |
| ex:myCar rdf:type ex:hasEngine . | I13 |
| ex:myCar rdf:type ex:Vehicle . | I14 |

Fig. 3. An example rule set and an example ontology

*3.2.1 First Phase:*

Three tasks are performed in sequence in the first phase: the construction of a sharable alpha network, the initial fact matching, and the collection of statistics. A *node sharing* mechanism is applied during the construction of the sharable alpha node to enable conditions with the same triple pattern to share the same alpha node. In the above example, rather than construct two separate alpha nodes for the condition (*?u* rdf:type *?v*) in the rule rdfs9 and the condition (*?x* rdf:type *?w*) in the rule rdfp15 as the original RETE algorithm does in figure 4a, only one alpha node (and only one alpha memory) is constructed in the sharable alpha network, as illustrated figure 4b. By sharing the alpha node, the total amount of memory allocated to these conditions can be cut to *1/n* if *n* conditions share a same alpha node.




(a) Original alpha network


(b) Sharable alpha network

Fig. 4. Alpha network constructed by both the original RETE algorithm and the two-phase RETE algorithm

Then, rather than continue to build the beta network as per the original RETE algorithm, RETE network construction is interrupted. At this stage a first iteration of the alpha network matching process is initiated and all alpha memories are populated. figure 5 gives the alpha network of the above example after the initial fact matching round. Statistics about the ontology, such as the number of matched facts for each condition, or join selectivity factor between two joining conditions and so on, can then be gathered (without introducing large amounts of extra processing as seen in a full a-priori execution). In this research the number of matched facts for each condition is gathered as an example of the statistics that could be performed at this stage.

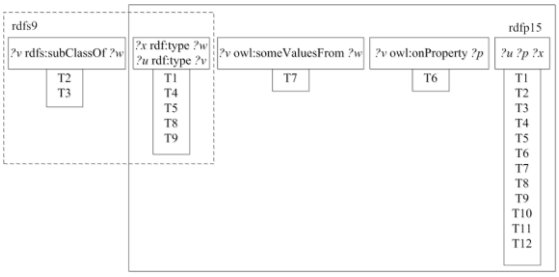



Fig. 5. The RETE network after the first phase

*3.2.2 Second Phase:*

The second phase of this algorithm builds an optimized beta network for the ontology using the statistics collected in the first phase, and then continues propagating facts (cached in the alpha memories) in the beta network as per the original RETE algorithm. Two heuristics from the state of the art, including a *most specific condition first* heuristic and a *pre-evaluation of join connectivity* heuristic, are incorporated to optimize the join sequences. However, rather than apply the heuristics statically based only on the rule set as described in section 2.4, the heuristics are applied dynamically taking into account the collected statistics about the particular ontology being reasoned over using the chosen rule set.

As discussed earlier the *most specific condition first* heuristic places the most specific conditions at the beginning of a join sequence and the least specific condition at the end to reduce the number of intermediate results. Here the number of matched facts for each condition element is used as the specificity for reordering join sequences. Figure 6 illustrates how it works following the above example. After the initial (alpha-network) fact matching, the join sequence of rdfs9 is already ordered with the most specific in the first and the least specific last and hence join sequence reordering is not applied to it. The rule rdfp15 however needs to swap the position of the third condition (?*x* rdf:type ?*w*) and the fourth condition (?*u* ?*p* ?*x*) as the former is less specific than the latter (the former has more matched facts (12 matched facts) than the latter (5 matched facts)). Join sequences before and after reordering are named differently as rdfp15 (before) and rdfp15 (after).

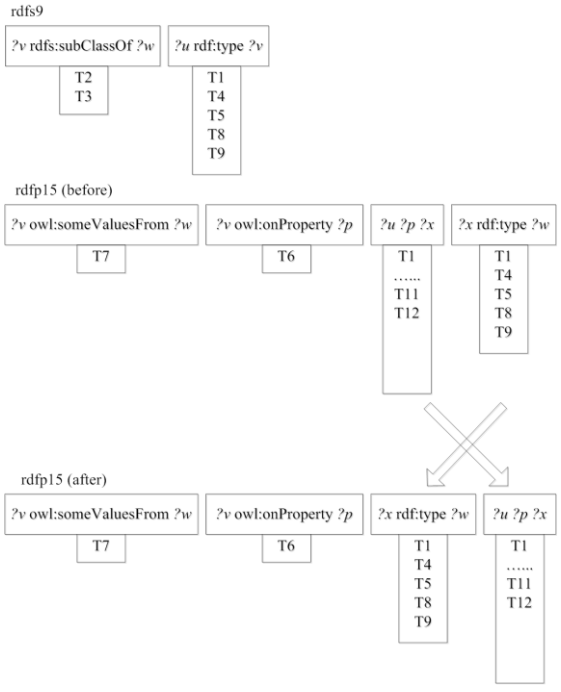



Fig. 6. An illustration of the most specific condition first heuristic in the two-phase RETE algorithm

The *pre-evaluation of join connectivity* heuristic ensures the join sequence is connected (i.e. where possible each condition should share a common variable with the conditions before it in the join sequence), which would otherwise lead to Cartesian product join and generate a large amount of beta network intermediate results. This is done here by scanning the join sequence from the left (front) to right (end), swapping each encountered unconnected condition with the first following connected condition in the join sequence. This can ensure the connectivity of the join sequence maintaining to the greatest extent possible the partial ordering introduced by the most specific condition first heuristic already applied to the join sequence.

Figure 7 exemplifies the *pre-evaluation of join connectivity* heuristic. Let $C_1$, $C_2$ … $C_{n-1}$ be a connected partial join sequence and together they form $C_{pre}$. Let $C_n$ be an unconnected condition. Therefore the heuristic searches from $C_{n+1}$ for the *first* condition connected to $C_{pre}$, which is $C_m$ in this example. Then $C_m$ is switched before $C_n$ and the remaining conditions (including $C_n$ and those following it) are shifted one place towards the end (as if shown by the second joins sequence in figure 7). As the join sequence has already been ordered by the most specific condition first heuristic, $C_m$ is then the most specific condition after $C_n$ that connects to $C_{pre}$. If no condition is found to connect to $C_{pre}$, then $C_n$ is left in place, and the check continues with $C_{n+1}$ in this case. As the rule rdfs9 and rdfp15 in our simple case are already connected the application of this heuristic does not change the join sequence.

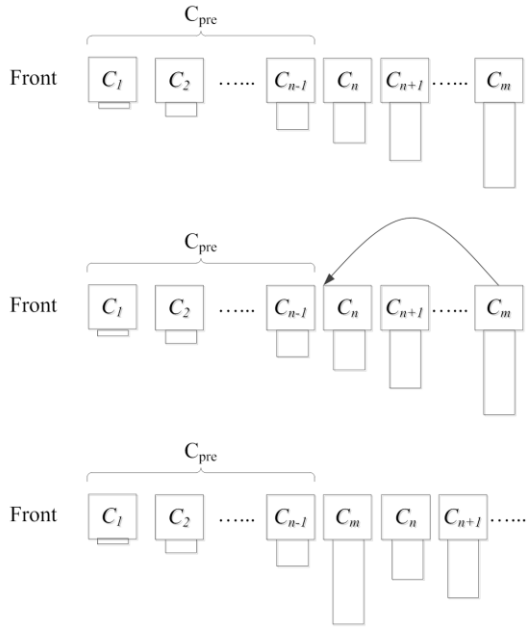
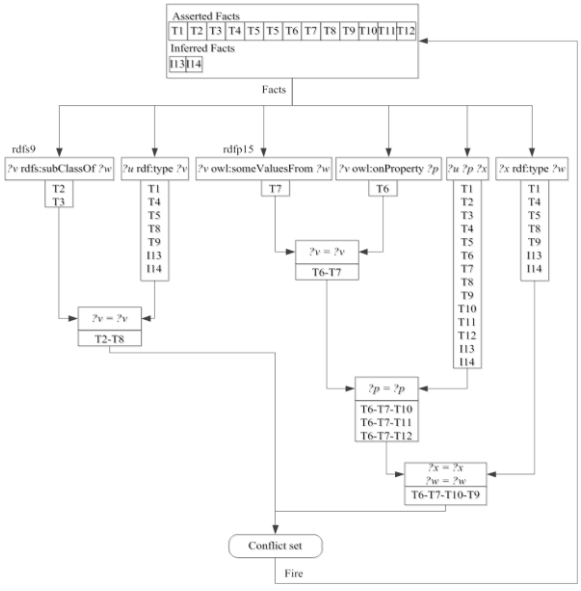
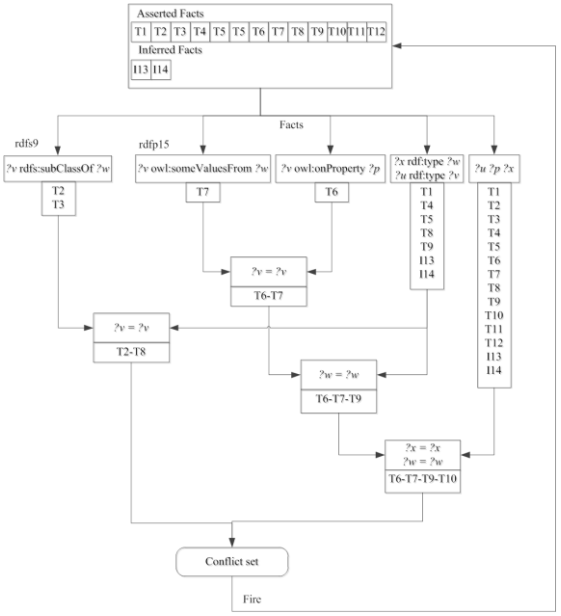


Fig. 7. An illustration of the pre-evaluation of join connectivity heuristic in the two-phase RETE algorithm.

The construction of a customized beta network then resumes based on the optimized join sequences. Facts stored in the alpha memory (because of the initial alpha network matching) continue to join in the beta network and to fire rules as per in normal RETE algorithm until no rules can be fired. Figure 8 continues with the above example and compares two RETE networks after all facts are matched: one is constructed by the normal RETE algorithm (figure 8a) and the other is constructed by the two-phase RETE algorithm (figure 8b). Intermediate results generated by join operations are listed in the box under the corresponding beta node. We can see from the diagram that the amount of facts cached in alpha memories reduces because a node sharing alpha network is constructed. In addition the re-ordering of the join sequence of rule rdfp15 causes a reduction of cached intermediate results in the beta network. It is important to note that the correctness of the algorithm is not affected and the resulting reasoned knowledge base is the same for both algorithms.




(a) The complete RETE network constructed by the normal RETE algorithm




(b) The complete RETE network constructed by the two-phase RETE algorithm

Fig. 8. The complete RETE network

### *3.3. Discussion*

In this section, merits of the composition algorithms are discussed in three respects: composition levels, semantics independence and automatic composition. Then limitations of the composition algorithms are also discussed. The discussion can give hints as to which composition algorithm is better for certain circumstances.

First, it is quite common for reasoners to work with a different rule set (e.g. a different set of OWL fragments or application-specific rules) so it is important that our approach remains semantic independent. Both of the composition algorithms are independent of a particular semantics and therefore can be applied for different rule sets (even generic application- rule sets). All steps involved in the two-phase RETE algorithm, i.e. the construction of the alpha network with the node sharing heuristic, the collection of statistics of the ontology, and finally the construction of the beta network with assistance from the join reordering heuristics and the collected statistics, operates only on different parts of the RETE network and hence it is apparent that they can work with a different semantics/rule set. The operation of the selective rule loading algorithm relies on the rule-construct dependencies of the used

reasoning semantics/rule set, nevertheless the rule-construct dependencies are not hardcoded in the algorithm and the way to analyze rules for rule-construct dependencies is designed to be independent of particular semantics. As can be seen later in section 4, the rule-construct dependencies of pD* (which is selected to demonstrate our proof-of-concept implementation) are implemented in a text file which can be easily swapped with other rule-construct dependencies when a different semantics, e.g. OWL 2 RL, is used. As a matter of fact, the rule-construct dependencies of all OWL 2 RL rule set have been analyzed in [43] and it shows OWL 2 RL can work well with the selective rule loading algorithm.

Second, the two composition algorithms compose at different levels: the selective rule loading algorithm at the rule set level and the two-phase RETE algorithm at the RETE algorithm level. At first glance, they are different in terms of the difficulties of implementation: the two-phase RETE algorithm requires refactoring of the internal RETE algorithm whilst the selective rule loading algorithm can be easily implemented as a standalone component without modifying the reasoning algorithm itself. In addition, since the selective rule loading algorithm is independent of reasoning algorithms, it has the potential to be used with other rule-based reasoners such as resolution-based reasoners whilst the two-phase RETE algorithm can only work on RETE-based rule-entailment reasoners. Both optimizations can be applied independently or together and do not interfere with each other.

Third, the two-phase RETE algorithm is a fully automatic composition approach requiring no manual analysis of either the rule set or the ontology. Although manual analysis of rules may be required for rule-construct dependencies (as a naïve approach which is easy to perform if the rule set is small and relative static), the application of the selective rule loading algorithm is automatically once rule-construct dependencies are constructed and the analysis for rule-construct dependencies is a once-off cost for each new rule set only and then the rule can be selectively loaded (automatic rule analysis can be also performed as long as the set of (OWL/domain-specific) constructs to be included in the rule-construct dependencies is provided). The selective rule loading algorithm can also be performed off-line manually, but this requires a large amount of manual analysis. As an aside, it is very unlikely that any domain-specific rules designed by an application domain expert would have already manually pre-optimized the ordering of condition elements based on any reasoning specific heuristic.

A limitation of the selective rule loading algorithm is that any (terminological) updates in the ontology after a rule set is selected may introduce OWL constructs into the ontology that were not previously present in the ontology at the rule selection stage. These new constructs might then require rules that were previously deselected and not loaded. This can be solved through re-executing the rule selection, where it should be possible to maintain cache results stored in the existing alpha and beta network nodes. However, for ease of implementation the simplest approach is a full re-reasoning of the ontology. Terminological update is not a problem for two-phase RETE algorithm as it loads the entire rule set. Hence changes in the ontology can be reflected immediately in the reasoning without re-executing the entire composition.

There are two potential problems worth examining for the two-phase RETE algorithm.

The first potential problem arises from the iterative fact matching process of RETE rule matching so the number of matched facts for any condition element collected from the initial (alpha) fact matching does not accurately represent the number of facts that eventually match that condition when RETE terminates. Hence it appears that the number of matched facts collected in the first phase of the two-phase RETE algorithm is only accurate for optimizing the first round of RETE fact matching. However, an investigation of the RETE network when reasoning over 19 ontologies (a summary of these ontologies can be found in the evaluation section) indicates that for most of these ontologies most reasoning work is done in the first RETE round: 15 of a total of 19 ontologies have an average of 75% joins performed in the first iteration and for the remaining 4 ontologies this percentage is still above 50%; furthermore an average of 83% inferred facts are generated in the first cycle. Hence it appears to be appropriate to optimize the RETE network by applying heuristics based on statistics collected only in the first phase.

The second potential problem, derived from the first potential problem above, concerns the effectiveness of using the number of matched (alpha) facts to order join sequences in general cases. While it may not be accurate to say that a condition with $n$

matched facts is always more specific than a condition with *n+1* matched facts as (1) these counts are only collected from the initial alpha network iteration, therefore it is very possible that the condition with n matched facts may have more matched triples in subsequent RETE cycles, and (2) the selectivity factor may vary for two conditions and it may be that the condition with less matched facts actually generates more intermediate results. This can be partially mitigated by introducing more sophisticated mechanisms for specificity estimation, for example exploiting more information such as the number of variables in condition elements, the cardinality of values to be joined, etc., all of which can be gathered before or in the first phase. In the current documented approach only the number of matched facts for each condition element is collected, but the presented approach is designed to support collecting more statistics and metrics. As described later, even this simplistic and crude approach substantially reduces memory and reasoning time.

## 4. COROR: A Composable Rule-entailment Owl Reasoner for Resource Constrained Devices

The reasoner composition algorithms described in section 3 were prototyped as a proof-of-concept resource-constrained reasoner named COROR. As COROR is only implemented to investigate if the reasoner composition approaches designed can reduce resource consumption for rule-entailment reasoning), the minimal design of COROR identifies three core components: a RETE engine, a framework for the loading/parsing/ storage/manipulation of ontologies, and the designed composition algorithms. Many other components of practical uses, such as a query interface or a remote reasoning interface and so on, are omitted for this implementation. However there should be no doubt that they can work well with our composition algorithms. A structural overview of COROR is given in figure 9.

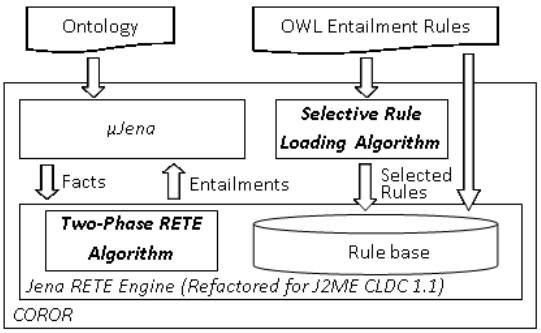



Fig. 9. An overview of COROR

The target platform for COROR is the Sun SPOT sensor platform v4.0 (blue)[1]. The hardware and software specifications of this platform are listed in Table 1. The choice of Sun SPOT sensor platform is motivated by two aspects: it is a representative of typical resource constrained devices and it has a full range of tools from software development using Java in Netbeans to software testing using a fully functional emulator.

Table 1. Specs of the Sun SPOT sensor platform

| | |
|---|---|
| CPU | 180MHz 32 bit ARM920T |
| RAM | 512KB |
| Flash | 4MB |
| OS | Squawk VM |
| API | J2ME CLDC 1.1 |

Rather than implementing every component in figure 9 from scratch, we chose to refactor and reuse some existing software. The µJena framework [49] is a J2ME version of Jena but without any reasoning capability. It was selected for ontology loading, parsing and storing. To add reasoning capability to µJena the RETE engine from the original desktop version of Jena was refactored to run on Sun SPOT. This involved extensive code-level refactoring from the J2SE to the J2ME CLDC 1.1 Java profile (in particular for container classes), and the identification and refactoring of reasoning affiliated classes (e.g. *RETEClauseFilter*, *RETEQueue*, *InfGraph*, and so on). The selection of Jena RETE engine was mainly driven by two reasons: (1) it has the maximum compatibility with µJena which means the least code refactoring is needed and, (2) as a typical rule-entailment reasoner implementing composition algorithms on Jena RETE can demonstrate the equal applicability of the

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

composition algorithms to other rule-entailment reasoners. Finally a sample Jena-compatible pD* rule set was authored. The final output was a resource-constrained OWL rule-entailment reasoner (without composition capabilities), which will then be referred as *enhanced μJena*.

The implementation of the reasoner composition algorithms was then based on enhanced μJena to implement COROR, an extension of enhanced *μ*Jena with both optimizations included. The selective rule loading algorithm was implemented as a standalone component, which runs before RETE engine starts, to generate a list of names of the selected rules to be loaded by the RETE engine. The analysis of the input ontology for contained OWL constructs is through naively enumerating all OWL constructs using the *listStatement* method provided by enhanced μJena. For example, in order to test the existence of owl:FunctionalProperty, a method invocation *listStatement*(null, null, owl:FunctionalProperty) is made. The rule-construct dependencies for our pD* rule set were manually analyzed (just once) and materialized in a local file. Each entry of this file represents the rule-construct dependency of a rule and an entry should look like (in a BNF-like notation):

***rule-name***′:′***semantic-level***′:[′***premises***′]

->[′***consequences***′]′

The usage of each field is apparent from its naming. Since pD* rules are used in this proof-of-concept implementation, semantic levels are hence RDFS, OWL-Lite and OWL-DL, corresponding to the semantics a pD* rule could belong to. This field is useful when rules need to be loaded according to the semantic level. Multiple premises and consequences are separated by commas. The *premises* field and the *consequences* fields are left blank if a rule does not have premise constructs or consequence constructs. The rule-construct dependencies file is loaded and parsed in memory before rule selection starts.

The difficulties of implementing the two-phase RETE algorithm mainly resided in two aspects: the enablement of node sharing in Jena RETE engine and the breaking down of Jena RETE algorithm into two parts to collect statistics about the ontology after the first phase. The node-sharing problem was mainly caused by the fact that Jena RETE uses rule-specific and position-based variable indexing and binding, which impedes the sharing of alpha nodes: the same condition at distinct positions in different rules has distinct binding vectors. This problem is solved in this implementation by using a *dual-vector* approach which stores the variable position information and the bound values separately in two vectors. Then the vector for bound values can then be shared among rules and the corresponding position information can be swapped in/out when the shared condition node is used for different rules.

To solve the second problem of breaking the RETE construction process into 2 phases, the construction of the alpha network and the beta network are split and wrapped into two separate Java methods. Join sequence of a rule is determined by the position each alpha node (*RETEClauseFilter*) in the join vector of the rule and hence the adjustment of join sequence turns out to be change the position of an alpha node in the join vector. For this proof-of-concept implementation only the number of matched triples for each alpha node is collected by counting the size of the vector associated to each alpha node, but different statistics can be collected and applied here.

Composition mechanisms are programmed to work individually or to work together in a hybrid way with a view to complementing each other to further preserve resources. Hence four COROR composition modes are derived, which are 1: *COROR-noncomposable (using only* enhanced *μ*Jena), 2: *COROR-selective*, 3: *COROR-two-phase,* and 4: *COROR-hybrid.* Each mode corresponds to using none, one, or two composition algorithms. This also enables us to evaluate the composition algorithms separately and together. In the hybrid mode the selective rule loading algorithm produces a selective rule set and then the two-phase RETE algorithm builds a customized RETE network according to the ontology to be reasoned over and the selective rule set.

The pD* semantics were implemented as 39 rules in Jena format (examples are given in figure 3), including 16 D* entailment rules and 23 P entailment rules. Four built-in functions were implemented for constraint checking or for assigning anonymous nodes to literals. They include isPLiteral() that checks if a variable is a plain literal, isDLiteral() that checks if a variable is a well formed datatype literal, assignAnon() that assigns an anonymous node to a variable, and notLiteral() that checks if a variable is not a literal. Clashes (XML-clash, part D-clash, P-clash, see more about clashes in [44]) are not handled in this implementation. This is consistent with assumptions

by some previous work that in a resource-constrained environment the consistency of ontology is either ensured by the ontology developer or checked offline by an ontology server [3, 38].

*Selecting the Rule Set: pD* vs OWL 2 RL*

OWL 2 has been standardized by W3C since 2009. A big improvement of OWL 2 from OWL 1 is that it is shipped with three language fragments each of which is designed to provide efficient reasoning for particular reasoning algorithms (e.g. rule-entailment), particular tasks (e.g. conjunctive query answering or classification) or particular expressiveness (e.g. EL++ [6]). OWL 2 RL [13] is one of the fragments and it is designed to provide scalable OWL 2 reasoning (PTIME combined complexity) on rule-entailment reasoners without sacrificing too much expressiveness. Hence it shows potential to be implemented in COROR. However, we still chose pD* for a number of reasons. First, there is not a publicly available off-the-shelf OWL 2 RL rule set and compared to pD* rules, the implementation of an efficient and error-free OWL 2 RL rule set is nontrivial as the performance of OWL 2 RL rules largely rely on how the list processing builtins are implemented. Second as discussed in section 3.3 the composition algorithms are designed to be independent of semantics and hence the implementation of pD* in COROR does not exclude the possibility of OWL 2 RL. As a matter of fact, section 3.5 of [43] has discussed in detail and established that OWL 2 RL can work with composition algorithms and rule-construct dependencies for OWL 2 RL rules are presented. Moreover, the aim of this research is focused on the design and evaluation of our composition algorithms rather than constructing an off-the-shelf resource-constrained reasoner. Hence, considering the difficulties involved in implementing an OWL 2 RL rule set for Jena and the small contribution at the time to the result of this work by providing an OWL 2 RL rule set for COROR, we opted for the pD* rule set. However considering the popularity of OWL 2 RL, we are investigating this as future work.

## 5. Evaluation

The purpose of the evaluation was to verify if composition algorithms could reduce resource consumption for rule-entailment OWL reasoning. For this purpose, our evaluation was designed to test and compare the performance of different COROR composition modes to reason over ontologies. Two things need to be clarified here: 1) the term reasoning refers to computing all entailments of an ontology according to the given rule set, and 2) the term performance refers to both time performance and memory performance.

### *5.1. Experiment Design and Execution*

Two experiments were designed. A first experiment was an *intra-reasoner comparison* to investigate the change of reasoning performance (i.e. time performance and memory performance) of a same rule-entailment reasoner with and without our composition algorithms. In our case, four COROR composition modes were compared side by side. A second experiment was an *inter-reasoner comparison* that compared our composable rule-entailment reasoner with some other rule-entailment reasoners from the state of the art to investigate if a composable rule-entailment reasoner would be more suitable (in terms of performance) than the off-the-shelf rule-entailment reasoners. For this experiment, COROR-hybrid was compared with four in-memory rule-entailment reasoners, which are Bossam 0.9b45, Jena v2.6.3 (with forward reasoning only, aka Jena-forward in this paper), BaseVISorv1.2.1, and swiftOWLIM v3.0.10. COROR-hybrid was used in this comparison because it combines both composition algorithms and therefore can represent the performance gain of both. The selected reasoners have similar semantics avoiding biases in the comparison that one reasoner is slower (respectively uses more memory) because it implements a lot more semantics than the other: SwiftOWLIM and BaseVISor adopted the pD* entailment with axiomatic triples and consistency rules; Jena was configured to use the same rule set as COROR; the expressivity of Bossam (closed source) was not defined on either its website[2], publications [24], or implementation, thus made it difficult to judge its inference capability. Pellet was also included to show how COROR performs compared to a full-fledged, complete DL reasoner. However since the expressivity of Pellet is superior to COROR using a pD* rule set, Pellet was only included to give readers an intuition of how COROR performs rather than to compare with COROR side by side. In fact none of the rule-entailment reasoners in the

[2] http://bossam.wordpress.com/

experiment can perform complete OWL-DL reasoning as performed by Pellet.

The memory and the time used by each reasoner to reason over ontologies were measured as metrics for the performance comparison. Reasoning time was measured by differentiate the time measured (through Java method *System.CurrentTimeMills()*) before and after reasoning was performed (for COROR, reasoning is performed by calling the method *InfGraph::prepare()*). Memory required for reasoning was measured by subtracting the free memory from the total memory (*Runtime.getRuntime().totalMemory()-Runtime.getRuntime().freeMemory()*)after the reasoning is performed. At this stage, all entailments are calculated, entailments materialised, and all caches are filled, so the RETE network reaches its maximum size and the memory usage measured at this moment is then the maximum memory usage of COROR.

The time and memory for reasoning were measured separately in different executions of COROR to reduce interference between their measurements. Each result of time/memory used in the evaluation is the average of 10 individual measurements to reduce the error in each measurement. Furthermore the Java garbage collection was performed by explicitly calling *System.gc()* 20 times before each memory measurement to release garbage so interference from non-recycled garbage memory can be reduced as much as possible. A threshold of 30 minutes was set to avoid excessively long reasoning. Any reasoning process still running after 30 minutes was manually terminated.

The intra-reasoner comparison was performed on the Sun SPOT sensor platform board emulator. It has a 180MHz processor, 512KB RAM and 4MB ROM as shown in Table 1. The inter-reasoner comparison was performed on desktop computer as the reasoners selected for comparison (written in J2SE) cannot run on Sun SPOT (running J2ME CLDC 1.1) and it is non-trivial to port them to a J2ME CLDC 1.1 platform. The environment where the inter-reasoner comparison was performed is: Eclipse Helios with Java SE 6 Update 14 with 128MB maximum heap size, Dual Core CPU @ 2.4GHz, 3.25GB RAM.

Ontologies used in the experiments are listed in Table 2. Eleven ontologies (Table 2a) were used in the intra-reasoner comparison and six more ontologies (Table 2b) were used in the inter-reasoner comparison. Their selection was based on three factors: (1) they were selected from different domains, which to some extent is able to represent the diversity of ontologies used in a resource-constrained environment, (2) they vary in expressivity avoiding unintentional biases that some OWL constructs are over- or under-used, and (3) they are well-known and commonly used so are relatively free from errors. URIs of the selected ontologies are given at the end of the paper.

Table 2: Ontologies used in the experiments

(a) Ontologies used in the intra-reasoner comparison

| Ontology | Expressivity | class/property/ individual | Size (triples) |
|---|---|---|---|
| teams | ALCIN | 9/3/3 | 87 |
| owls-profile | ALCHIOF(D) | 54/68/13 | 116 |
| Koala | ALCON(D) | 20/7/6 | 147 |
| university | SIOF(D) | 30/12/4 | 169 |
| Beer | ALHI(D) | 51/15/9 | 173 |
| mindswapper | ALCHIF(D) | 49/73/126 | 437 |
| Foaf | ALCHIF(D) | 17/69/0 | 503 |
| mad_cows | ALCHOIN(D) | 54/17/13 | 521 |
| Biopax | ALCHF(D) | 28/50/0 | 633 |
| Food | ALCOF | 65/10/57 | 924 |
| mini-tambis | ALCN | 183/44/0 | 1080 |

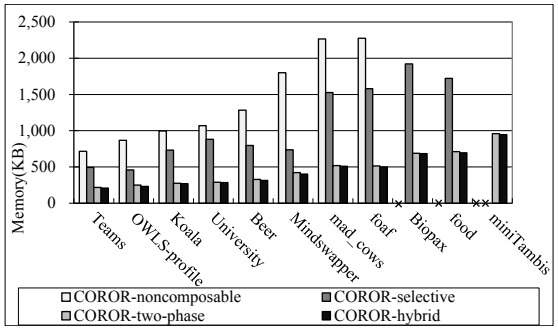
(b) Six more ontologies used in the inter-reasoner comparison

| Ontology | Expressivity | class/property/ individual | Size (triples) |
|---|---|---|---|
| amino-acid | SHOF(D) | 55/24/1 | 1465 |
| atk-portal | ALCHIOF(D) | 169/147/75 | 1499 |
| Wine | SHOIN(D) | 77/16/161 | 1833 |
| Pizza | ALCF(D) | 87/30/0 | 1867 |
| tambis-full | SHIN | 395/100/0 | 3884 |
| Nato | ALCF(D) | 194/885/0 | 5924 |

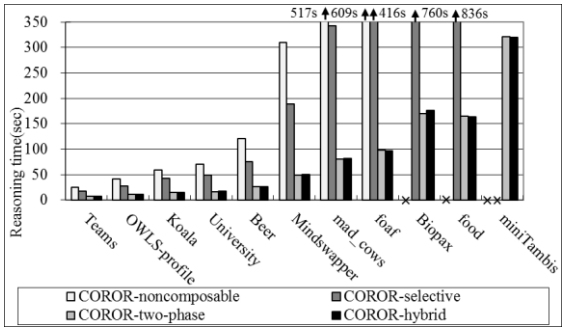
### *5.2. Results and Discussions*

*Intra-reasoner comparison*

The memory and the time required by different COROR composition modes to reason over the selected ontologies are respectively given in figure 10a and figure 10b. Results are presented in an ascending order according to the size of the ontology.




(a) Memory performance




(b) Time performance

Fig. 10. Results of the intra-reasoner comparison

For all tested ontologies, a composable COROR (i.e. COROR with the composition algorithms turned on) greatly outperforms the noncomposable version of COROR (i.e. COROR with an original RETE engine) in terms of both memory and time. COROR-selective uses on average 35% less memory than COROR-noncomposable (with a std. dev. as 13%) and on average 33% less time (with a std. dev. as 4%). COROR-hybrid uses on average 74% less memory than COROR-noncomposable (with a std. dev. as 3%) and on average 78% less time (with a std. dev. as 6%). COROR-hybrid uses on average 75% less memory than COROR-noncomposable (with a std. dev. as 3%) and on average 78% less time (with a std. dev. as 6%). For 8 ontologies (i.e. *teams, owls-profile, koala, university, beer, mindswappers, mad_cows,* and *foaf*) COROR-hybrid requires less than 512KB memory (RAM size of Sun SPOT), COROR-two-phase can reason 6 ontologies within 512KB RAM, COROR-selective can reason the 2 smallest ontologies within 512KB RAM, but COROR-noncomposable is unable to reason any of the ontologies within 512KB RAM. Whenever the reasoning process can not be achieved only in RAM extensive paging of RAM state to/from (Flash) storage is required. Some results for some ontologies are missing (marked as an "X") as COROR cannot finish the reasoning within the given time threshold due to excessive paging and processing, which required manual termination of the reasoning process, i.e. COROR-noncomposable failed to reason *biopax*, *food* and *miniTambis* even after 30 minutes.

However, the big differences between the memory/time saving of COROR-two-phase and that of COROR-selective indicate that the two composition algorithms vary in their capability to reduce resource consumption. To investigate this difference, we performed an in-depth examination of the RETE networks constructed by all four COROR modes. Generally match operations and join operations dominate the running time of a RETE algorithm. We chose the amount of match operations (#M) and the amount of join operations (#J) as respective indicators for the amount of time spent in the alpha network and the amount of time spent in the beta network. Similarly, the amount of intermediate results generated and cached in the network (#IR) was selected as an indicator for the memory usage of COROR RETE engine, and hence the amount of intermediate results generated and cached in alpha network ($\#IR_\alpha$) and beta network ($\#IR_\beta$) are therefore respective indicators for the memory usage of the alpha network and the beta network.

By observing these indicators, we noticed that compared with COROR-noncomposable COROR-two-phase had less #M and $\#IR_\alpha$ but had almost the same #J and $\#IR_\beta$. This implies that the memory gain and the time gain of COROR-selective comes mainly from the not constructing of the alpha nodes for the deselected rules but beta network benefitted little from the deselected rules (as #J and $\#IR_\beta$ remained almost unchanged). Further investigation of the rules de-selected by COROR-selective indicates that they were already manually optimized by the original pD* rule author with conditions ordered for optimal join sequences. Hence in an execution of COROR-noncomposable, even when these rules that would be deselected in the corresponding execution of COROR-selective are loaded into the RETE engine, they still perform very few join operations leading to a nearly empty beta network. For this reason, for the tested ontologies, deselecting these rules in executions of COROR-

selective did not achieve substantial memory or time improvement in beta network.

The same set of manually optimized pD* rules was also used for COROR-two-phase however experiment results in figure 10 yield interesting results in terms of performance gain that the application of COROR-two-phase had much more performance gain than that of COROR-selective. We then investigated the RETE network generated by COROR-two-phase. It revealed a similar situation that most performance gain came from the construction of a shared alpha network but the join sequence reordering heuristics had little effect on reducing #J and $\#IR_\beta$. As a matter of fact, only two rules, rdfp11 and rdfp15, had their join sequences reordered. For rdfp11, the first two conditions in the join sequence swapped position, leading to no change in #J and $\#IR_\beta$. For rdfp15, the last two conditions in the join sequence switched position according to the *most specific condition first* heuristic, leading to a small reduction of #J and $\#IR_\beta$. However, as the rule rdfp15 accounted for a very small portion of the total #IR in the whole RETE network (only 0.91% for the *wine* ontology and even smaller for the other tested ontology), the performance gain from the reordering join sequence of this rule is subtle.

From the above analysis, we can learn that the different performance gains of COROR-selective and COROR-two-phase is because of the construction of a shared alpha network in the two-phase RETE algorithm. In fact, conditions are highly shared among pD* rules, e.g. the condition (*?v* owl:sameAs *?w*) is shared by rdfp6, rdfp7, rdfp9, rdfp10 and rdfp11 and the wildcard condition (*?v ?p ?l*) is shared by 21 rules. As only one alpha node is constructed for each shared condition, the more conditions shared among rules and also the more rules share a condition, the less $\#J/\#IR_\beta$ per rule and hence the more performance gain from a shared alpha network. On the other hand, this also implies that a sharable alpha network can lead to more performance gain in alpha network than merely removing a part of the alpha network if a rule set with highly shared conditions is used (given that semantics are not impaired). Since COROR-hybrid is only a simple add-up of the selective rule loading algorithm and the two-phase RETE algorithm, the above analysis also explains the performance gain between COROR-two-phase and COROR-hybrid (figure 10).

To establish that the heuristics introduced in the second phase of the two-phase RETE algorithm can lead to performance gain, we deliberately changed the join sequence of three rules, i.e. rdfp1, rdfp2, and rdfp4, to make them less optimized, but maintaining the exact same semantics. Modified rules are respectively named rdfp1m, rdfp2m and rdfp4m (figure 11). The performance of COROR-two-phase and COROR-noncomposable were measured again using the same settings as those in figure 10 (on the same Sun SPOT emulator with the same set of ontologies) but with both (original and modified) rule sets. Results are given in figure 12. The word "*original*" and "*modified*" are affixed to the corresponding COROR mode to distinguish if the original or the modified rule set is used.

It is apparent from figure 12a and figure 12b that COROR-noncomposable-modified used a lot more memory and time than COROR-noncomposable-original to reason over a same ontology. Compared with the results presented in figure 10, four more ontologies, i.e. *Beer, Mindswapper, mad_cows,* and *foaf,* cannot be reasoned over by COROR-noncomposable-modified within the time threshold, which indicates the poor performance to run a set of less optimized rules in a noncomposable reasoner. However, the COROR-two-phase-modified had very close performance as COROR-two-phase-original. Inspection into the RETE networks showed that the join sequences of the modified rules were reordered by the heuristics in the second phase of the two-phase RETE algorithm to the same join sequences as they were in the original rule set. This demonstrates that even the crude and simplistic heuristic used in the fully automated join reordering process in COROR yields results at least as performant as rule sets manually reordered by rule-authoring specialists, with the added advantage that the reordering applied in COROR is also ontology-specific. Sub-optimal join ordering would be common in manually authored application-specific rules. Moreover, compared to the reduction of #M and $\#IR_\alpha$ in COROR-two-phase-modified, the reduction of #J and $\#IR_\beta$ in COROR-two-phase-modified was very substantial therefore indicating that the majority performance gain comes from the join sequence reordering in the second phase of the two-phase RETE algorithm.

**Original Rules**
[rdfp1: (*?p* rdf:type owl:FunctionalProperty), (*?u ?p ?v*), (*?u ?p ?w*), notLiteral(*?v*)
→ (*?v* owl:sameAs *?w*)]

[rdfp2: (*?p* rdf:type owl:InverseFunctionalProperty), (*?u ?p ?w*), (*?v ?p ?w*) → (*?u* owl:sameAs*?v*)]

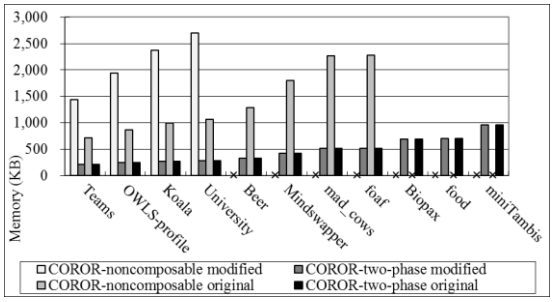
[rdfp4: (*?p* rdf:type owl:TransitiveProperty), (*?u ?p ?v*), (*?v ?p ?w*) →(*?u ?p ?w*)]

**Modified Rules**
[rdfp1m: (*?u ?p ?v*), (*?u ?p ?w*), (*?p* rdf:type owl:FunctionalProperty), notLiteral(*?v*)
→ (*?v* owl:sameAs *?w*)]

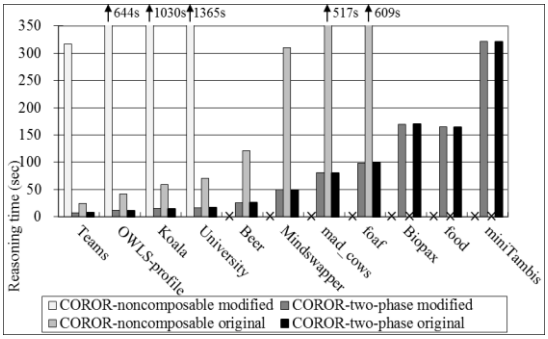
[rdfp2m: (*?u ?p ?w*), (*?v ?p ?w*), (*?p* rdf:type owl:InverseFunctionalProperty)
→(*?u* owl:sameAs *?v*)]

[rdfp4m: (*?u ?p ?v*), (*?v ?p ?w*), (*?p* rdf:type owl:TransitiveProperty) →(*?u ?p ?w*)]

Fig. 11. Original and modified version of the rule rdfp1, rdfp2 and rdfp4




(a) Memory performance




(b) Time performance

Fig. 12. Reasoning performance of COROR-two-phase and COROR-noncomposable for both modified and original rule set

Based on the above discussion, it can be inferred that the two-phase RETE algorithm would be better at handling a rule set with long and sub-optimal join sequences and with highly shared conditions whilst the selective rule loading algorithm would work better on rules with shorter join sequences (no more than 2 conditions) or where most conditions are not shared among rules.

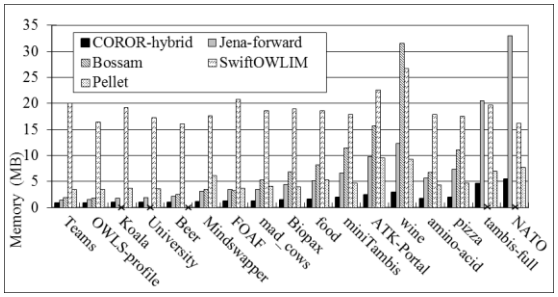
*Inter-reasoner comparison*

Results of the inter-reasoner comparison are given in figure 13 (memory performance in figure 13a and time performance in figure 13b). As shown in the figures, the time performance of COROR-hybrid is comparable to Jena-forward and BaseVISor. However, in contrast to the results of the intra-reasoner comparison presented in figure 10 where COROR-hybrid used much less reasoning time than COROR-noncomposable (where the same RETE engine as the one used by Jena-forward but refactored to run on J2ME platform is used, as described in section 4), COROR-hybrid only slightly outperforms Jena-forward in the inter-reasoner comparison. Considering the modifications made to port Jena RETE engine to Sun SPOT (as described in section 4), we infer that the major reason causing this difference is that the J2SE container classes used in Jena-forward are much more optimized than the their naïve counterparts implemented by the authors and the µJena authors.

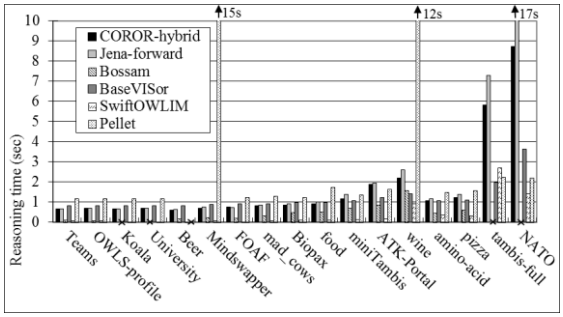
SwiftOWLIM was the fastest reasoner for most ontologies in the inter-reasoner comparison (except for *Tambis-full* where BaseVISor was the fastest). Bossam is also fast for many ontologies. For smaller ontologies such as *Teams*, *OWLS-profile, Beer,* Bossam can compete with swiftOWLIM, however it failed with errors for four ontologies including *Koala, University, tambis-full and NATO* all of which are successfully reasoned over by the other reasoners. Pellet generally had quite constant performance for most selected ontologies regardless of their sizes. However for smaller ontologies it used more time than all the other rule-entailment reasoners. Pellet used much more time to reason over *wine* and *mindswapper*. This is because the very expressive structures used in the terminology (which in the Wine ontology were specifically designed to stress the reasoner) slowed the tableau-based reasoning process. Pellet did not have results for the *Beer* ontology as inconsistencies were detected.

In the experiment the memory performance of COROR was much better than the other reasoners. It used the least memory among all reasoners to reason over all tested ontologies. This was especially the

case for the smaller selected ontologies. The memory usage of Bossam and Jena-forward grew much faster than that of COROR-hybrid as the size and the complexity of the ontology increased. For example, for the *ATK-portal* ontology Bossam used 6 times more memory than COROR and for the *Wine* ontology it used 13 times more memory than COROR. The memory footprint for swiftOWLIM was much larger than COROR even for the very small ontologies, e.g. for the *teams* ontology it used 20MB of memory which is 15 times larger than COROR. This shows swiftOWLIM trades increased memory usage for time improvements. BaseVISor hides its reasoning process from external inspection so it was not possible to accurately measure its memory usage, and therefore it was omitted from the memory comparison. The overall results indicate that a much smaller memory footprint is needed for COROR to reason over small-/medium-sized ontologies without sacrificing time performance. This demonstrates that from a performance perspective it is feasible to run in a resource-constrained environment.




(a) Memory performance




(b) Time performance

Fig. 13. Results of the inter-reasoner comparison

## 6. Conclusion

There are many reasons why some semantic reasoning should be performed at or close to edge-nodes in a semantic network of nodes, such as improved fault-tolerance, reduced centralized processing load, reduced raw-data traffic and routing overhead, more localized decision making and aggregation, faster reaction in time-critical scenarios, supporting larger distributed knowledge-bases, etc, and they all could contribute to a robust system in critical situations. In most cases edge-nodes are more resource-constrained, and in an extreme case of a semantic sensor network, nodes are extremely low-specification. To enable semantic reasoning in such cases, resource-efficient semantic reasoning needs to be designed for resource-constrained environments. This research investigates the use of reasoner composition to reduce the resource consumption of rule-entailment reasoning. Two algorithms, a selective rule loading algorithm and a two-phase RETE algorithm, are designed to compose an optimized reasoning process depending on the rule set and reasoning algorithm, without sacrificing the generality or semantics supported in the reasoner. A performance evaluation of a proof-of-concept implementation of the algorithms indicates that reasoner composition using just the rule set can save on average 35% of the reasoning memory and 33% of the reasoning time, and composition based on the reasoning algorithm can save on average 74% of the reasoning memory and 78% of the reasoning time. A comparison between our implementation and other rule-entailment reasoners from the state of the art (including a mobile reasoner) shows that our implementation uses much less memory than the others (with similar semantics) without compromising on time. The results show reasoner composition could be considered a pre-requisite to enable rule-entailment semantic reasoning in resource-constrained environments.

## Acknowledgement

This work is supported by the Irish Government in "Network Embedded Systems" (NEMBES), under the Higher Education Authority's Program for Research in Third Level Institutions (PRTLI) cycle 4.

## Reference


[1] A. Acciarri, D. Calvanese, and G. D. Giacomo, “QUONTO: querying ontologies,” in Proceedings of the 20th national conference on artificial intelligence, 2005, pp. 2-3.

[2] W. Tai, J. Keeney, and D. O’Sullivan, “COROR: A COmposable Rule-entailment Owl Reasoner for Resource-Constrained Devices,” in Proceedings of The 5th International Symposium on Rules: Research Based and Industry Focused (RuleML’11), 2011, pp. 212–226.

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

[4] P. Wang, G. J. Zheng, L. Fu, E. Patton, T. Lebo, L. Ding, Q. Liu, J. Luciano, and D. McGuinness, “A semantic portal for next generation monitoring systems,” Proceedings of the 10th International Semantic Web Conference (ISWC’11), pp. 253–268, 2011.

[5] C. Forgy, "Rete: A Fast Algorithm for the many pattern/many object pattern match problem”, Artificial Intelligence, Volume 19, Issue 1, Pages 17-37, 1982.

[6] F. Baader, S. Brandt, and C. Lutz, “Pushing the EL envelope,” in Proceedings of the 19th International Joint Conference on Artificial Intelligence, 2005, vol. 29, no. 8.

[7] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, Eds., the Description Logic Handbook: Theory, Implementation and Applications, 2nd ed. Cambridge, UK: Cambridge University Press, 2007.

[8] P. Baumgartner, “Hyper Tableau — The Next Generation,” in International Conference on Automated Reasoning with Analytic Tableau and Related Methods (TABLEAU’98), 1998, no. x, pp. 60-76.

[9] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov, “OWLIM : A family of scalable semantic repositories,” Semantic Web Journal, vol. 2, no. 1, pp. 33-42, 2011.

[10] M. D’Aquin, A. Nikolov, and E. Motta, “How much semantic data on small devices?,” in Proceedings of the International Conference on Knowledge Engineering and Management by the Masses, 2010, pp. 565–575.

[11] Y. Kazakov, M. Krötzsch, and F. Simančík, “Concurrent Classification of EL Ontologies,” in Proceedings of International Semantic Web Conference, 2011, pp. 305–320.

[12] J. J. Carroll, D. Reynolds, I. Dickinson, A. Seaborne, C. Dollin, and K. Wilkinson, “Jena: Implementing the Semantic Web Recommendations,” in Proceedings of the 13th International World Wide Web Conference, 2004, pp. 74-83.

[13] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz, “OWL 2 Web Ontology Language Profiles,” W3C Recommendation, 2009. [Online]. Available: http://www.w3.org/TR/owl2-profiles/.

[14] C. Choi, I. Park, S. J. Hyun, D. Lee, and D. H. Sim, “MiRE: A Minimal Rule Engine for context-aware mobile devices,” in Proceedings of International Conference on Digital Information Management, 2008, pp. 172-177.

[15] P. Desai, C. Henson, P. Anatharam, and A. Sheth, “Demonstration : SECURE -- Semantics Empowered resCUe Environment,” in Proceedings of the 4th International Workshop on Semantic Sensor Network, 2011, pp. 115–118.

[16] U. Hustadt, B. Motik, and U. Sattler, “Reducing SHIQ-description logic to disjunctive datalog programs,” in Proceedings of the 9th International Conference on Knowledge Representation and Reasoning (KR2004), 2004, pp. 152–162.

[17] “SPIN –SPARQL Inference Notation,” 2012. [Online]. Available: http://spinrdf.org/.

[18] P. Barnaghi, W. Wang, C. Henson, and K. Taylor, “Semantics for the Internet of Things : early progress and back to the future,” International Journal on Semantic Web and Information Systems, vol. 8, no. 1, pp. 1–21, 2012.

[19] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker, “Description Logic Programs : Combining Logic Programs with Description Logic Categories and Subject Descriptors,” in Proceedings of International Conference on World Wide Web, 2003, pp. 48-57.

[20] S. Hawke, “Surnia,” 2003. [Online]. Available: http://www.w3.org/2003/08/surnia/.

[21] I. Horrocks and P. F. Patel-Schneider, "Reducing OWL entailment to description logic satisfiability”, Journal of Web Semantics, Volume 1, Issue 4, Pages 345-357, 2004.

[22] T. Ishida, “Optimizing rules in production system programs,” in Proceedings of the 7th National Conference on Artificial Intelligence (AAAI’87), 1988, pp. 699-704.

[23] T. Ishida, “An Optimization Algorithm for Production Systems,” IEEE Transactions on Knowledge and Data Engineering, vol. 6, no. 4, pp. 549-558, 1994.

[24] M. Jang and J.-C. Sohn, “Bossam : An Extended Rule Engine for OWL Inferencing,” in Proceedings of International Workshop on Rules and Rule Markup Languages for the Semantic Web, 2004, pp. 128-138.

[25] “Hoolet reasoner,” 2012. [Online]. Available: http://owl.man.ac.uk/hoolet/.

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

[27] S. Hasan, E. Curry, M. Banduk, and S. O’Riain, “Toward Situation Awareness for the Semantic Sensor Web : Complex Event Processing with Dynamic Linked Data Enrichment,” in Proceedings of the 4th International Workshop on Semantic Sensor Network, 2011, pp. 69–81.

[28] C. Matheus, K. Baclawski, and M. Kokar, “BaseVISor: A Triples-Based Inference Engine Outfitted to Process RuleML and R-Entailment Rules,” In Proceedings of International Conference on Rules and Rule Markup Languages for the Semantic Web, pp. 67-74, Nov. 2006.

[29] G. Meditskos and N. Bassiliades, “DLEJena : A Practical Forward-Chaining OWL 2 RL Reasoner Combining Jena and Pellet,” Journal of Web Semantics, vol. 8, no. 1, pp. 1-12, 2009.

[30] J. Mendez and B. Suntisrivaraporn, “Reintroducing CEL as an OWL 2 EL Reasoner,” Proceedings of the 2009 International Workshop on Description Logics, 2009.

[31] “The Gene Ontology,” 2012. [Online]. Available: http://www.geneontology.org/.

[32] T. Özacar, Ö. Öztürk, and M. O. Ünalir, “Optimizing a Rete-based Inference Engine using a Hybrid Heuristic and Pyramid based Indexes on Ontological Data,” Journal of Computers, vol. 2, no. 4, pp. 41-48, 2007.

[33] P. Ruch, J. Gobeill, C. Lovis, and A. Geissbühler, “Automatic medical encoding with SNOMED categories,” BMC medical informatics and decision making, vol. 8, no. 1, Jan. 2008.

[34] C. Seitz and R. Schönfelder, “Rule-based OWL reasoning for specific embedded devices,” in Proceedings of International Semantic Web Conference, 2011, pp. 237–252.

[35] T. Kawamura and A. Ohsuga, “Toward an ecosystem of LOD in the field: LOD content generation and its consuming service,” in Proceedings of the 11th International Semantic Web Conference (ISWC’11), 2012, pp. 98–113.

[36] O. Younis, S. Fahmy, and P. Santi, “An Architecture for Robust Sensor Network Communications,” International Journal of Distributed Sensor Networks, vol. 1, no. 3–4, pp. 305–327, 2005.
[37] A. Sinner and T. Kleemann, “KRHyper - In Your Pocket System Description,” in Proceedings of International Conference on Automated Deduction, 2005, pp. 452-457.
[38] L. A. Steller, S. Krishnaswamy, and M. M. Gaber, “Enabling scalable semantic reasoning for mobile services,” International Journal on Semantic Web & Information Systems, vol. 5, no. 2, pp. 91–116, 2009.
[39] M. Stocker and M. Smith, “Owlgres: A Scalable OWL Reasoner,” in Proceedings of the 5th International Workshop on OWL: Experiences and Directions, 2008, vol. 432.
[40] V. Vassiliadis, J. Wielemaker, and C. Mungall, “Processing OWL2 ontologies using Thea : An application of logic programming,” Processing, vol. 2009, no. Owled, 2009.
[41] R. Volz, S. Decker, and D. Oberle, “Bubo - Implementing OWL in rule-based systems,” 2003. [Online]. Available: http://www.daml.org/listarchive/joint-committee/att-1254/01-bubo.pdf.
[42] “Witmate, the only one pure java Logic/Rule Engine executable from J2ME to J2SE/J2EE.” [Online]. Available: http://www.witmate.com/default.html.
[43] W. Tai, “Automatic Reasoner Composition and Selection,” PhD Thesis, School of Computer Science and Statistics, Trinity College Dublin, 2012.
[44] 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, no. 2-3, pp. 79-115, Oct. 2005.
[45] D. Tsarkov and I. Horrocks, “FaCT ++ Description Logic Reasoner : System Description,” in Proceedings of the International Joint Conference on Automated Reasoning (IJCAR 2006), 2006, pp. 292-297.
[46] Y.-wang Wang and E. N. Hanson, “A Performance Comparison of the Rete and TREAT Algorithms for Testing Database Rule Conditions,” in Proceedings of the 8th International Conference on Data Engineering (ICDE’92), 1992, pp. 88-97.
[47] Y. Zou and T. Finin, “F-owl: An inference engine for semantic web,” in Workshop on Formal Approaches to Agent-Based Systems, 2004, pp. 238-248.
[48] J. Zhou, L. Ma, Q. Liu, L. Zhang, Y. Yu, and Y. Pan, “Minerva : A Scalable OWL Ontology Storage and Inference System,” in Proceedings of the 2006 Asian Semantic Web Conference (ASWC06), 2006, pp. 429-433.
[49] F. Crivellaro, G. Genovese, and G. Orsi, “µJena - Software.” [Online]. Available: http://poseidon.ws.dei.polimi.it/ca/?page_id=59. [Accessed: Mar-2012].
[50] D. J. Scales, “Efficient matching algorithms for the SOAR/OPS5 production system,” Technical Report KSL-86-47, Department of Computer Science, Stanford University, 1986.
[51] A. Romero, B. Grau, and I. Horrocks, “Modular combination of reasoners for ontology classification,” Proceedings of the 2012 Description Logics Workshop (DL’12), vol. 846 CEUR, 2012.
[52] G. Meditskos and N. Bassiliades, “A Rule-Based Object-Oriented OWL Reasoner,” IEEE Transactions on Knowledge and Data Engineering, vol. 20, no. 3, pp. 397–410, 2008.
[53] 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, vol. 5, no. 2, pp. 51–53, Jun. 2007.
[54] “RacerPro 2.0,” 2012. [Online]. Available: http://www.racer-systems.com/products/racerpro/preview/index.phtml.
[55] B. Motik, R. Shearer, and I. Horrocks, “Hypertableau reasoning for description logics,” Journal of Artificial Intelligence Research, vol. 36, pp. 165–228, 2009.
[Food] Food ontology, http://www.w3.org/2001/sw/WebOnt/guide-src/food
[FOAF] Friend of a Friend ontology, http://xmlns.com/foaf/0.1/
[Beer] Beer ontology, http://www.purl.org/net/ontology/beer
[Biopax-Level1] Biopax level 1, http://www.biopax.org/release/biopax-level1.owl
[Koala] Koala ontology, http://protege.stanford.edu/plugins/owl/owl-library/koala.owl
[Madcows] mad cows ontology, http://www.cs.man.ac.uk/~horrocks/OWL/Ontologies/mad_cows.owl
[Mindswappers] mindswappers ontology, http://www.mindswap.org/2003/owl/mindswap
[Minitambis] mini tambis ontology, http://www.mindswap.org/ontologies/debugging/miniTambis.owl
[Owls profile] Owls profile, http://www.daml.org/services/owl-s/1.1/Profile.owl
[Teams] teams ontology, http://owl.man.ac.uk/2005/sssw/teams
[University] university ontology, http://www.mindswap.org/ontologies/debugging/university.owl
[Amino acid] amino acid ontology, http://www.co-ode.org/ontologies/amino-acid/2005/10/11/amino-acid.owl
[NATO] NATO ontology, http://www.mindswap.org/ontologies/IEDMv1.0.owl
[MGED] MGED ontology, http://mged.sourceforge.net/ontologies/MGEDOntology.daml
[Pizza] Pizza ontology, http://www.co-ode.org/ontologies/pizza/pizza_20041007.owl
[AKT-portal] AKT Portal ontology, http://www.aktors.org/ontology/portal
[Tambis full] Tambis full ontology, http://www.mindswap.org/ontologies/tambis-full.owl
[Tap] Tap ontology, http://www.aktors.org/ontology/portal
[Wine] Wine ontology, http://www.w3.org/2001/sw/WebOnt/guide-src/wine