Resource-Constrained Reasoning Using a Reasoner Composition Approach

Editor(s): Christophe Guéret, Data Archiving and Networked Services (DANS), KNAW, Netherlands; Stephane Boyera, World Web Foundation, France; Mike Powell, IKM Emergent, USA; Martin Murillo, Data Connectivity Initiative - IEEE
Solicited review(s): Christophe Guéret, Data Archiving and Networked Services (DANS), KNAW, Netherlands; Aidan Hogan, Universidad de Chile, Chile; Christophe Dupriez, DESTIN SSEB, Belgium

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: To increase the interoperability and accessibility of data in sensor-rich systems, there has been a recent proliferation of the use of Semantic Web technologies in sensor-rich systems. Quite a range of such applications have emerged, such as hazard monitoring and rescue, context-aware computing, environmental monitoring, field studies, internet of things, and so on. These systems often assume a centralized paradigm for data processing, which does not always hold in reality especially when the systems are deployed in a hostile environment. At runtime, the infrastructure of systems deployed in such an environment is also prone to interference or damage, causing part of the infrastructure to have limited network connection or even to be detached from the rest. A solution to such a problem would be to push the intelligence, such as semantic reasoning, down to the device layer. A key enabler for such a solution is to run semantic reasoning on resource-constrained devices. This paper shows how reasoner composition (i.e. to automatically adjust a reasoning approach to preserve only a "well-suited" amount of reasoning for a given ontology) can achieve resource-efficient semantic reasoning. Two novel reasoner composition algorithms are introduced and implemented. Evaluation indicates that the reasoner composition algorithms greatly reduce the resources required for OWL reasoning, potentially facilitating greater semantic reasoning on sensor devices.

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

1. Introduction

In the past decade, there has been an increasing interest in applying Semantic Web technologies to sensor-rich systems to enhance the interoperability of heterogeneous data, to bring about more intelligent data processing, and to make data more easily accessible on the Web. Much of this research has been developed for a wide range of uses, 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 common approach to integrate Semantic Web technologies into Sensor Network is to use a centralized paradigm: sensor data is sent over the network to a comparatively powerful central server (either a local gateway [15] or implemented as remote services on the Web [4, 35, 27]) where all semantic processing takes place, such as the conversion of sensor data into RDF, semantic-based data aggregation, semantic query answering, visualization, and so on. This approach has been proven to work well for cases with reliable network connections and a relatively stable environment, on the other hand it may not always be the best option for some harsher use cases where some sensor-rich systems, such as forest fire monitoring, seafloor monitoring or space exploration, need to be deployed. Harsh surroundings often mean that the deployment of an onsite powerful server with fast and stable network connections could be very costly or even on occasions technically impossible. At runtime, the infrastructure of such systems is also prone to interference or damage, causing a limited network connection or even a complete detachment of some parts of the infrastructure. In some other cases, to cover a large geographical area a system might comprise of hundreds and thousands of sensors of various types. The overhead required by such systems to relay raw un-processed data can be very high, and furthermore centralized semantic data processing could act as a bottleneck in such systems and incur significant overhead.

Such environments often require a sensor-rich system to be robust enough to retain part (or all) of data processing on the networked devices. Thus for

semantic-enabled sensor-rich systems that are of interest in this work, semantic data processing such as semantic uplifting of sensor data, data aggregation, and semantic-assisted decisions/actuations will be required to be performed on network edge devices such as sensors or on sensor cluster heads [36]. In this way, more meaningful aggregated data can be sent to a central server, which on the one hand can reduce the amount of data that is required to be transmitted over a network and on the other hand can achieve local autonomy to some extent when the central server is out of reach. As a key enabling step for on-device semantic data processing, it is essential to push semantic reasoning towards the edge of the system where various resource-constrained devices (for data gathering, actuation, and data aggregation) often reside. In the extreme case they may be only sensors with very limited resources.

Existing semantic reasoners are too resource-intensive to be directly ported on resource-constrained devices [3, 34, 38]. A study in [10] shows that a desktop rule-entailment reasoner can take from tens to several hundred KBs of memory (depending on characteristics of the ontology) to reason each RDF triple loaded into memory. Hence although technically portable to a small device with some code-level modifications, a desktop semantic reasoner can still easily drain the resources of 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).

In this work, we introduce the idea of *reasoner composition* and also describe how reasoning composition can achieve more resource-efficient semantic reasoning for resource-constrained devices. The rationale behind reasoner composition is that the different expressiveness of ontologies presents different requirements for reasoning capabilities. A simple example would be that reasoning over a simple RDFS ontology only needs some of the RDFS rules whereas reasoning over a more expressive OWL ontology would probably need the entire OWL 2 RL ruleset. This difference of expressiveness of ontologies could then be exploited to tailor the semantic reasoning process such that only the required reasoning abilities are loaded and run.

Different semantic reasoning approaches have been developed in the past decades and they are widely adopted by state of the art semantic reasoners. This work focuses only on the composition of rule-based reasoners, in particular rule-entailment reasoners, mainly due to the long standing history of using forward-chaining rule to handle domain semantics in sensors-rich systems, and also the potential in composing a rule-based reasoner [2]. Section 2.1 presents the semantic reasoning approaches and also a justification for focusing this research on rule-based reasoners. In our research, two novel reasoner composition algorithms have been devised to compose a rule-entailment reasoner from two complementary perspectives: A *selective rule loading* algorithm that composes a selective ruleset according to the expressiveness of the ontology to be reasoned over, and a *two-phase RETE* algorithm that adjusts the RETE rule matching algorithm to operate in a way tailored for the particular ontology to be reasoned over. Both reasoner composition algorithms were implemented in a proof-of-concept resource-constrained semantic reasoner (called COROR). Experiment results indicate that a composable COROR uses significantly less resources than a non-composable reasoner, thus potentially facilitating semantic reasoning in resource-constrained environments.

The paper is organized as follows. Section 2 presents background and related work. Section 3 elaborates the design of the two composition algorithms. Implementation details for a prototype composable reasoner ("COROR") are provided in section 4, followed by Section 5 describing an evaluation of the approach. Section 6 concludes with a discussion of the significance, limitations of the work and directions for future work of the research.

2. Background and Related Work

This section first presents a categorization of semantic reasoning approaches and then justifies the choice of rule-entailment reasoners as the focus of the composition research. This categorization enables the composition research to be carried out for a type of reasoner rather than for a particular reasoner implementation. After finding that rule-entailment reasoners are the most suitable type for composition, the RETE algorithm, as the most widely used rule matching algorithm, is then described. Finally, related work is discussed, including resource-constrained ontology reasoners, query optimizations and scalable rule-based ontology reasoning.

2.1. Semantic Reasoning Approaches

Quite different semantic reasoning approaches have been developed in the past decades. A survey of reasoning approaches was carried out earlier in this research over a set of 26 state of the art semantic reasoners. The survey suggests that their reasoning approaches can be categorized into five categories: *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 established OWL entailments to DL satisfiability checking based on semantic connections between the two [21]. Normally a DL tableau reasoner translates an OWL ontology into a DL knowledge base, and then does the satisfiability checking using well-established DL tableau algorithms. This basically tries to construct a non-contradictory model for the DL knowledge base using a set of consistency preserving *transformation rules*. If such a model is found, then the DL knowledge base is said to be *satisfiable*, or otherwise *unsatisfiable*. State of the art reasoners falling into this category include FaCT++ [45], Pellet [53], RacerPro [54], and HermiT [55].

Rule-entailment reasoners compute total entailments for OWL ontologies based on a set of forward-chaining production rules that (partially) implement an OWL semantic profile. These rules are often termed semantic *entailment rules*.

Depending on the way OWL semantics are interpreted in entailment rules, this work distinguishes *direct style* entailment rules (DS rules) from *RDFS style* entailment rules (RS rules). DS rules are often dynamically transformed from ontological assertions according to OWL direct semantics [60]. For example, a terminological assertion (*Car owl:subClassOf Vehicle*) is transformed to a DS rule $Car(x) \rightarrow Vehicle(x)$. O-DEVICE [52] and DLEJena [29] use DS rules. The Description Logic Programs (DLP) perform a similar transformation between Description Logic and Logic Programs [19]. RS rules are derived from RDF compatible OWL semantic conditions [60]. Unlike DS rules in which OWL semantics are implied, OWL constructs are explicitly used in RS rules. An example RS rule that does a similar reasoning as the above example would look like:

$$(?c\ rdfs{:}subClassOf\ ?d) \wedge (?x\ rdf{:}type\ ?c) \rightarrow (?x\ rdf{:}type\ ?d)$$

pD* rules [44] and OWL 2 RL entailment rules [17] follow the RDFS style. DS rules are mainly used for reasoning assertional data, but RS rules can reason both terminological and assertional data. Rule-entailment reasoners using RS rules include BaseVISor [28], OWLIM [9], Bossam [24], MiRE4OWL [26], the Jena forward chaining engine [12] and EYE[1].

At the heart of **resolution-based reasoners** is a resolution-based logic engine. Therefore, the reasoning process for resolution-based reasoners first normally translates an OWL ontology into logic programs consisting of sentences written in the supported logic language, such as Prolog [40, 41], Datalog [16], or first-order TPTP (Thousands of Problems for Theorem Provers) language [25]. It then performs ontology reasoning using a corresponding logic engine. Both RS and DS rules are also used by resolution-based reasoners, in order to implement OWL semantics. However, resolution-based reasoners differ from rule-entailment reasoners, in that here rules are matched in a backward-chaining manner using a resolution engine rather than a forward-chaining manner using production engines. F-OWL [47], Surnia [20], Thea [40] and the Jena backward-chaining engine [12] are resolution-based reasoners using RS rules. Resolution-based reasoners using DS rules include KAON2 [56] and Bubo [41]. Hoolet [25] translates an ontology into first-order TPTP axioms and performs reasoning using the Vampire automated theorem prover.

Some other reasoners have been devised to provide efficient reasoning for a particular OWL profile or for some specific purpose or task, and they use a dedicated reasoning approach. These reasoners are classified as **miscellaneous reasoners** in our categorization. For example, CEL [30] and ELK [11] implement polynomial time classification algorithms designed for OWL 2 EL [13]. One item worth noting is that in ELK the term "composition" in "composition/decomposition optimization" is interpreted differently from this work: it means to bring together atomic concepts into complex subsumption axioms. Other miscellaneous reasoners include Owlgres [39] and QuOnto [1] that use query rewriting algorithms to efficiently answer queries over large datasets falling into OWL 2 QL expressivity [13].The SPIN approach uses SPARQL

[1] http://eulersharp.sourceforge.net/

CONSTRUCT and SPARQL UPDATE to handle OWL 2 RL reasoning.

Some reasoners combine two or more of the above reasoning approaches to exploit the advantages of each approach. These reasoners are categorized as **hybrid reasoners**. Example hybrid reasoners are Minerva that combines DL tableau approach with resolution-based approach [48], DLEJena that combines DL tableau approach with a rule-entailment approach [29], Jena that combines rule-entailment approach with a resolution-based approach [12], and Pellet that incorporates a forward-chaining rule engine for rule handling. Reasoning in Stardog [2] is performed by explicitly separating the terminological part of the ontology from its assertional part. The terminological part is reasoned using Pellet, and the assertional reasoning is performed at query-time using query-rewriting informed by the reasoned terminological part of the knowledge base. The reasoning approach presented in [51] falls into this category as well. Rather than using a (comparatively) slow but expressive reasoner to reason over the ontology with an expressive DL language, this work extracts modules of the ontology each of which has an expressiveness falling into a lesser DL language that can be reasoned using less expressive but more efficient DL reasoners. An expressive reasoner is only used for a complete reasoning over the partial results generated by component reasoners.

A similar categorization of semantic reasoners can also be found in [47], which defines three categories: *DL tableau* reasoners, *full FOL* reasoners and *partial FOL* reasoners. This categorization has its own uses but it was not fine-grained enough for our reasoner composition research. Their categorization was based on underlying logics implemented by reasoners and it does not distinguish reasoners using different reasoning algorithms. Such algorithmic differences are quite important for reasoner composition research, as the different reasoning algorithms can support different composition approaches.

Choice of Rule-entailment reasoners

The choice of rule-entailment reasoners as the target of our composition research is based on consideration of: (1) the difficulty for a composition process of using any of the other categories of reasoning approach, and (2) the suitability to operate in a resource-constrained environment.

First, the loose coupling characteristic of entailment rules would indicate that rule-entailment reasoners is a good candidate for a composability process than the other reasoning approaches.

The fact that RS rules often implement semantics in a fine-grained way and that each RS rule can be applied independently without other RS rules, indicates that RS rules could be more natively composable, based on the amount of semantics in the target ontology. DS rules are generated by parsing the ontology to be reasoned over according to rule templates. Therefore only appropriate but very specific rules are generated. However, DS rules are mainly used for reasoning assertional data, and terminological reasoning still relies on other reasoning approaches such as the DL tableau approach or RS rules, which makes it less appealing for reasoner composition research.

Transformation rules are used in tableau-based approaches for deciding the satisfiability of a DL knowledge base. However, as transformation rules describe procedures of DL tableau calculi, de-selecting transformation rules may impair the completeness of the original DL tableau calculus [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 achieve more efficient practical DL reasoning by DL tableau calculi, a wide range of complex, code-level optimizations are entwined into DL tableau implementations, such as different backtracking mechanisms, loop detection mechanisms, and model caching techniques [7]. To comply with a composition algorithm, these optimizations would also need to be adjusted at runtime, which would further complicate the possibility of dynamic composition of DL tableau reasoners.

Reasoners that belong to the miscellaneous category 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 hardwired algorithms make them less appealing as candidates for composability studies. In addition, the composability of a hybrid reasoner relies on each of the individual component reasoning approaches used, where composability for each approach has already been discussed above.

[2] http://stardog.com/

As a result of the considerations outlined in the discussion above, the basis for our composition research was narrowed down to rule-entailment reasoners and resolution-based reasoners, which are the two categories using entailment rules.

The suitability of applying both reasoning approaches to resource-constrained environments is discussed next.

In addition to OWL reasoning, a sensor-rich system often needs to react to events. For this purpose, domain-specific rules are often required. Compared to goal-directed resolution-based reasoners, rule-entailment reasoners use production engines. They have the intrinsic capability to receive events and in addition to trigger a corresponding non-semantic rule when a particular (combination of) event happens. 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]. As a matter of fact, some rule-entailment reasoners have already been implemented for resource-constrained environments [24, 26]. Such implementations further confirm the suitability of our choice of rule-entailment reasoners for our work.

As discussed above, the way DS rules are generated can compose an entailment ruleset with "well-suited" semantics for the assertional data of an ontology. Terminological reasoning and domain-specific reasoning still rely on RS rules. Considering that these two types of reasoning are important in sensor-rich systems, we chose a rule-entailment approach using RS rules to be the target of our composition research.

In this work, our research is not limited to rule composition but also the composition of a well-optimized reasoning algorithm. RETE is a fast pattern matching algorithm for forward-chaining production systems [5]. It forms the underlying rule matching algorithm for many general-purposed rule engines such as Drools and JESS[3]. RETE is also used as the underlying reasoning algorithm for most rule-entailment reasoners including the Jena forward engine, BaseVISor, Bossam, MiRE4OWL and O-DEVICE. Therefore in this work, the composability of RETE algorithm is studied. The remainder of this section provides readers with more knowledge on the RETE algorithm.

Although varied between implementations, a RETE algorithm in general performs rule matching using a data structure termed *RETE network*. Normally a RETE network comprises two parts. The *alpha* part consists of inter-connected alpha nodes each of which is responsible for testing if a fact matches against an *attribute* of a rule *condition*. Facts *matched* to one alpha node are passed to the next one for more attributive tests. Facts successfully matched to all attributes of a rule condition is said to match to the condition and are cached in the *alpha memory* for the condition. A fact stored in alpha memory is often termed a *token* or a *partial instantiation* of the rule. The *beta* part of a RETE network consists of networked beta nodes for finding *full instantiations* of a rule by *joining* tokens. Each beta node performs a *join* operation between two inputs each of which receives tokens from an alpha node or an upper-level beta node. Tokens received from either input are also cached in a separate *beta memory*. Successfully joined tokens are combined into a new (larger) token containing fields from both input tokens, and the new token is then sent to a lower-level beta node for further joins. Normally a beta network for a rule has its first beta node join between tokens from the first two alpha memories (they keep the facts matched to the first two conditions of a rule). The following beta nodes join between the previous beta node and the next alpha memory in the condition sequence. Outputs from the last beta node of a join tree are then full instantiations of a rule. Full instantiations are sent a *RETE terminal* node where actions associated with the rules are executed, also known as *rule firing*. Normally, a RETE engine does rule firing in iterations: it would find all possible full instantiations for facts in the current working memory, schedule a proper rule firing sequence, fire rules one by one in sequence, insert newly inferred facts into the working memory, and start another match-fire iteration based on the updated working memory. This match-fire iteration stops when an *inference closure* is reached for all rules, namely no more new facts can be inferred. In this work each such match-firing iteration is called a *RETE cycle* and all RETE cycles together are called a *RETE reasoning*.

The RETE algorithm used in this research is based on the Jena implementation and therefore is a variation of the one described above. One difference is that the alpha network of the RETE algorithm is an array of alpha nodes rather than a network. In each alpha node are three attributive tests each of which is either a URI test or a variable binding

[3] Drools: http://www.jboss.org/drools/
Jess: http://herzberg.ca.sandia.gov/

corresponding to a test of the subject, property or object position of a triple pattern. Although not using a network of alpha nodes, the term "alpha network" is kept in this work to conform to the RETE naming convention. Another difference from the original RETE approach is that action(s) of an entailment rule is the insertion of triples into working memory and therefore an action is also represented as a triple pattern in this work.

Before the RETE algorithm used in this research is formally described, notations are defined. Let U denote the set of URI constants, B the set of blank nodes, V the set of variables, L the set of literals, and $T = U \cup V \cup B$ the set of terms. Let $TP = \{(s,p,o)|s \in U \cup B, p \in U, o \in U \cup B \cup L\}$ denote the set of triples and $TPN = \{(s,p,o)|s \in U \cup V, p \in U \cup V, o \in U \cup V \cup L\}$ the set of triple patterns. Let a tuple $r = (CDT, ACT)$ denote an entailment rule where $CDT = \{tpn_i \in TPN\}$ is a sequence of conditions and $ACT = \{tpn_j \in TPN\}$ a sequence of consequences. The notation $r.CDT.i$ is used to identify the ith condition of rule r and similarly $r.ACT.i$ for the ith consequence of rule r. R is used in this work to denote a set of entailment rules.

Now we define match operation, join operation, alpha memory and beta memory. Given $tp \in TP$ and $tpn \in TPN$, then

$$match(tp, tpn) = \begin{cases} TRUE, & (tp.s = tpn.s \vee tpn.s \in V) \wedge \\ & (tp.p = tpn.p \vee tpn.p \in V) \wedge \\ & (tp.o = tpn.o \vee tpn.o \in V) \wedge \\ FALSE, & otherwise \end{cases}$$

is a function executing over a pair of a triple *tp* and a triple pattern *tpn*. In short, it returns TRUE when *tp* instantiates *tpn*, and FALSE otherwise.

Let $WM = \{tp|tp \in TP\}$ denote the working memory, *tokenize* a function turning a triple into a token, an alpha node for a particular rule condition *tpn*, written as α^{tpn}, is then defined as

$$\alpha^{tpn} = \{tokenize(tp)|match(tp, tpn) \wedge tp \in WM \wedge tpn \in r.CDT \wedge r \in R\}$$

Based on alpha memories, the ith beta node of rule r, written as $\beta^{r.i}$, is defined recursively as

$$\beta^{r.i} = \begin{cases} \alpha^{r.CDT.i} \bowtie \alpha^{r.CDT.(i+1)}, & i = 1 \\ \beta^{r.(i-1)} \bowtie \alpha^{r.CDT.(i+1)}, & 2 \le i \le |r.CDT| - 1 \end{cases}$$

where $\bowtie$ represents the join symbol. The above definition captures the idea described in the beginning of this section: a beta node takes two inputs, performs pair-wise checks for consistent variable bindings for each input token, and generates a new token for each pairs with consistent variable bindings.

Now RETE cycle and RETE reasoning are defined. Let $O = \{\text{tp}|\text{tp} \in \text{TP}\}$ be an ontology, *fire* a function that generates inferred triples for each full instantiation found in each rule, a RETE reasoning $\Omega = \{(\text{A}_\text{k}, \text{B}_\text{k}, \text{WM}_\text{k})|1 \le k \le l\}$ is defined as a sequence of tuples and each tuple $(\text{A}_\text{k}, \text{B}_\text{k}, \text{WM}_\text{k})$ in this sequence is a RETE cycle. A_k, B_k and WM_k are respectively the alpha network, the beta network and the working memory in the kth RETE cycle. They are defined as

$$\text{A}_\text{k} = \left\{\alpha_k^{r.CDT.i} \middle| 1 < i < |r.CDT| \wedge r \in R\right\}$$

$$\text{B}_\text{k} = \{\beta_k^{r.i}|1 < i < |r.CDT| - 1 \wedge r \in R\}$$

$$\text{WM}_\text{k} = \begin{cases} O, & k = 1 \\ WM_{k-1} \cup fire\left(\beta_{k-1}^{r.(|r.CDT|-1)}, r.ACT\right), & 2 \le k \le l \wedge r \in R \end{cases}$$

The operand $\beta_{k-1}^{r.(|r.CDT|-1)}$ of *fire* represents the last beta memory for rule r at RETE cycle k-1. When WM_{k-1}=WM_k, the RETE algorithm terminates. Let $\Omega_\text{k} \in \Omega$ be a RETE cycle. A RETE reasoning Ω is then a chain of RETE cycles

$$\Omega_1 \to \cdots \to \Omega_l$$

where $\text{WM}_{l-1} = \text{WM}_l$.

RETE Optimization

In addition to ruleset composition, this research also focuses on automatically composing an appropriate RETE network for the particular ontology to be reasoned. Towards this goal, RETE optimizations are discussed in this section as background knowledge.

Caching tokens avoids the same match operations and join operations being performed again for the same tokens. It can greatly speed up the rule matching process. However the downside is the potential inflation of the beta network when join sequences are inappropriately arranged, which sometimes leads to a vast amount of cached tokens. An extreme case would be that two condition elements have no common variables, leading to Cartesian production joins and hence to a drastic

increase in the memory for storing tokens. To cope with this problem, different optimization heuristics have been designed to re-order RETE join sequences, but the common aim is to perform the most discriminating joins first.

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]. Briefly, the former heuristic is based on the rule of thumb that the more specific a rule condition is the more likely it has fewer matched facts. Accordingly pushing a more specific condition towards the front of a join sequence is more likely to generate fewer tokens in beta network [46, 32]. In practice, there are three criteria for evaluating the specificity of a condition [32]: (1) the more variables a condition has the less specific it is, (2) the more complex property a condition has the more specific it is, and (3) the less facts in the alpha memory for a condition when a RETE reasoning *ends* the more specific the condition is. As will be discussed in section 3.2, this research opts to use the third criteria. The pre-evaluation of join connectivity heuristic deems that a condition should have at least one common variable with previous conditions where possible. Therefore, it re-orders join sequences to avoid Cartesian product joins as much as possible. Cartesian product joins, if any, are then pushed to the end of the join sequence, where fewer input tokens are.

A direct application of RETE optimization heuristics considers only rules and therefore often fails to construct a customized join sequence for the particular fact base to be reasoned over [22, 23]. The work in [22, 23] shows that by taking characteristics of the fact base (ontology in this work) into account a more optimal RETE network can be constructed. However the approach proposed in [22, 23] involves the computation of a complicated cost model on a large amount of enumerated join sequences, which in our case is not quite suitable given the very limited computational resources. In section 3.2 a composition algorithm that composes a reasoner using the above two RETE optimizations is presented.

2.3 Related Work

This section discusses work related to this research including resource-constrained ontology reasoners, join optimization, and large-scale rule-based reasoning.

2.3.1 Resource-constrained Ontology Reasoners

Previous work has been devoted to resource-constrained OWL reasoners [3, 24, 26, 37, 38] and some optimizations are incorporated to reduce resource consumption, especially the memory footprint. This section describes some of the resource-constrained OWL reasoners and optimizations.

Mire4OWL [26] is designed to run on smart mobile phones for context-aware computing. It implements the rule-entailment approach and RS rules. To avoid an ever-growing fact base, it keeps only the latest context facts and makes extensive use of indexing to speed up duplication checking [14, 26]. μOR [3] is designed to help realize ambient intelligent medical devices. It uses dynamically generated DS rules for reasoning over assertional data and uses RS rules for reasoning over terminological data. A simple forward-chaining pattern matching algorithm is designed for rules matching. Experiments show that μOR only needs a small amount of memory and time to reason over a very small ontology (100 – 300 triples). Bossam [24] was originally designed to be a desktop reasoner but it has been implemented to run on J2ME CDC compatible mobile devices. Bossam follows the rule-entailment approach and RS rules. Based on our review of related literature, there is no evidence that it uses optimizations to reduce resource consumption for resource-constrained devices. mTableaux [38] uses a DL tableau calculi optimized for running on mobile devices. mTableaux uses a similar approach to ruleset composition to reduce resource consumption: It customizes the application of transformation rules according to the ontology but with a compromise on reasoning completeness. Pocket KRHyper [37] performs DL reasoning using a FOL theorem prover and hyper tableau calculi [8]. Two optimizations are employed to reduce resource consumption, including (1) use different DL/FOL transformation schemes for different parts of the knowledge base and (2) use distinct treatments for interests and disinterests. Both optimizations compose the reasoner according to characteristics of the knowledge base, and therefore could be considered to follow a reasoner composition approach.

None of the rule-entailment reasoners use any composition approaches. However, as both Mire4OWL and Bossam adopt the rule-entailment reasoning approach and RS rules, they should work well with the reasoner composition approaches

designed in this work. Similarly, the terminological reasoning in μOR also uses RS rules and should allow our reasoner composition approaches to be applied.

2.3.2 Query Optimizations

The RETE algorithm and query evaluation share some common characteristics: (1) They both need to partition records (or fact base) according to conditions specified in queries (or rules); (2) They both need to join partitions in order to find consistent variable bindings. These common characteristics show potential for the adoption of query optimization approaches for RETE optimization. Considering that join sequence reordering is a major type of RETE optimization and also a key part of the reasoner composition algorithms proposed in this research, this section focuses on query optimization techniques related to ordering join sequences.

Work in [56] shows that a poorly ordered join sequence can give rise to considerable degradation of query performance. Much of the research approach this issue by first enumerating query execution plans using dynamic programming and then finding an appropriate one according to cost models [57]. A similar approach is also used in RETE join sequence optimization [22, 23] but it uses a full a-priori RETE execution for gathering statistics about the fact base. The evaluation of a cost model is performed after the a-priori RETE execution.

Cardinality estimation is an idea that is commonly used for join sequence ordering in query optimization. Histogram approaches are commonly employed for such a purpose [59, 57]. However, [58] points out that as semantic constraints are often imposed upon RDF data, a single bucket histogram approach is often harnessed in a way that could lead to a mis-estimation of cardinality. The authors propose to use a characteristic set for cardinality estimation for RDF data. A characteristic set for an entity is the set of predicates that appears in the same triples in the data set. This approach is based on the observation that an RDF entity can usually be identified by only a subset of their emitting edges (or properties). This approach computes the number of distinct entities in the characteristic set and for each entity the number of occurrences of each of its emitting edge. These statistics are annotated in the characteristic set, and based on these statistics, the cardinality for triple pattern(s) is then calculated.

However the iterative characteristic of the RETE algorithm makes it difficult to produce an optimal RETE network by directly applying query optimization approaches. This is because compared to a single-round query evaluation, each RETE iteration will infer new facts, altering the dataset and therefore possibly offsetting already applied query optimizations. Nevertheless, as will be discussed in Section 3.3.2, optimizing only the first RETE cycle is sufficient in our scenarios as for the selected ontologies (they could represent the characteristics of the ontologies to be used in our scenario) the majority of RETE match and RETE join operations are performed in the first RETE iteration.

2.3.3 Scalable Rule-based Ontology Reasoning

Much effort has been spent on scaling rule-based reasoning to very large input by minimising the amount of data stored in memory. Similar approaches could also be applicable where there is a low availability of memory, as manifested by resource-constrained devices.

It is often deemed that the transitive, reflexive and symmetric semantics of *owl:sameAs* can cause considerable inflation in the number of inferred triples [9, 61]. One approach to such an issue is to use a single entity as a representative of a group of identical entities [9]. The research in [61] uses an identify-and-merge approach to build large sameAs cliques in a (comparatively) small memory. It loads sameAs assertions from disk by batch, merges them as partial cliques in memory and then appends the partial cliques to cliques back in disk. Considering the small-sized ontologies that could be used in resource-constrained devices, our research concentrates on pure in-memory reasoning.

The SOAR reasoning system [63, 62] has incorporated a range of optimizations to achieve scalable ontology reasoning. It uses partial-indexing to store and index only terminological data in memory. Assertional data is however left on secondary storage. DS rules are used in SOAR for calculating ontology entailments. To reduce the number of rules, rules with the same condition set are merged. In addition rules are indexed based on different forms of triple patterns to enhance the speed for looking up the right rules for a triple. Optimizations in SOAR system are not quite applicable in our research. The RS rules (pD* rules) used in our research are static and no two pD* rules share the exact same set of conditions. Hence the rule merging approach is not applicable. In addition

given the small number of pD* rules, indexing rules may lead to overhead offsetting the benefit gained from indexing.

A less related topic is parallel OWL inference [64, 61, 65]. The work in [64] proposes an approach to partition the assertional data of an ontology so that all data partitions can be reasoned by RDFS rules in parallel without impairing the soundness and completeness. The work in [65] describes WebPIE, a semantic reasoner using MapReduce for large scale rule-entailment reasoning [65]. It incorporates several optimizations to speed up reasoning, including loading schema triples in memory, data pre-processing to avoid duplicates and ordering the application of RDFS rules. Whereas parallel reasoning is not directly related to our research in that our target is to investigate the composability of a reasoner on a single device, research work on parallel reasoning still shows potential for resource-constrained reasoning on devices with multiple processors or distributed over multiple networked devices.

3. Rule Entailment Reasoner Composition

This section presents the design of two composition algorithms that tailor rule-entailment reasoning according to the particular ontology to be reasoned over, so as to enable resource-efficient ontology reasoning on resource-constrained devices. The pD* semantics [44] is used in this work to illustrate reasoner composition algorithms. The pD* semantics extends RDFS with some OWL features, including (in)equality, property characteristics, property restrictions, *disjointWith*, and *hasValue*. Semantics of these OWL features are partially supported in pD*. Other OWL features such as cardinality, Boolean combinations of class expressions, and *oneOf* are not included. Given the low resource availability on resource-constrained devices, the target ontology should not be too complicated and hence pD* semantics should provide sufficient semantic support. In addition, the pD* semantics is represented as a set of RS rules (often termed *pD* rules*). Each rule implements a small part of the semantics, facilitating ruleset composability. Furthermore, the pD* semantics has low entailment complexity: having PTIME entailment complexity without variables in the ontology and NPTIME entailment complexity with variables included.

3.1. Selective Rule Loading Algorithm

The selective rule loading algorithm dimensions a selected ruleset according to the amount of semantics contained in the target ontology. There are two motivating factors behind selective rule loading algorithm: (1) the diversity in the amount of semantics included in different ontologies, ranging from lightweight taxonomies using (partial) RDFS to ontologies with complicated concept descriptions using full OWL 1 or OWL 2 expressiveness [6, 33], and (2) the often very fine-grained semantics carried by each RS entailment rule. Therefore, by avoiding the loading of unnecessary rules, a reasoner can construct a smaller RETE network and hence can reduce memory consumption and reasoning time.

The selective rule loading algorithm employs what we call *rule-construct dependencies* to describe the use, as well as the generation of a construct in a rule. Each entry in rule-construct dependencies describes the terms from a given vocabulary $\mathcal{V}$, where $\mathcal{V} \subset \mathrm{U}$, are required for a rule r to fire and which terms are generated when the rule fires. The former type of terms is called *premise* terms of rule r, written as $P_r = \{t | t \in \mathcal{V}\}$, and the latter *consequent* terms of rule r, written as $C_r = \{t | t \in \mathcal{V}\}$. For pD* rules, $\mathcal{V}$ is therefore the pD* vocabulary (refer to [44] for the set of pD* vocabulary). Based on the above definitions, each entry of the rule-construct dependencies set should look like

$$P_r \xrightarrow{r} C_r$$

For example, let $?v \in V$, $?u \in V$, and $?w \in V$ be rule variables (to clarify, in the remaining part of this paper rule variables are denoted in this way), then the rule-construct dependency for rule *rdfs9*

$$\frac{(?v\ \textit{rdfs:subClassOf}\ ?w) \wedge (?u\ \textit{rdf:type}\ ?v)}{(?u\ \textit{rdf:type}\ ?w)}$$

is therefore

$$\{rdfs{:}\,subClassOf, rdf{:}\,type\} \xrightarrow{rdfs9} \{rdf{:}\,type\}$$

A full list of all rule-construct dependencies for pD* rules can be found in [43].

Before the selective rule loading algorithm is described in a more formal manner, some notations are defined. Here we use O to denote the ontology to be reasoned, R the set of entailment rules, and O^{R+} the entailment closure of O reasoned by R. From a

RETE's perspective, O^{R+} contains the same set of facts as what included in WM_j where j is the last RETE cycle. Let K_o be the set of terms from the vocabulary $\mathcal{V}$ and that appears in O, we then define K_O^{R+} to be the set of terms from $\mathcal{V}$ and also appearing in O^{R+}. We term K_O^{R+} a *R-closure* of K_o. From the definitions we can infer that a rule $r \in R$ will be fired in the reasoning of O only if $P_r \subseteq K_O^{R+}$. A corollary would be that only if each premise term of a rule exists in the original target ontology or as a consequent term of other rules, can the rule be considered as a candidate rule to be selected. In other words, if some premises of r do not appear in K_O^{R+} then the rule will not be fired and hence there is no need to load it.

Based on the above analysis, the selective rule loading is therefore a calculation of a *selective ruleset* $R_O^- = R - \{r | r \in R \wedge \exists t: (t \in P_r \wedge t \notin K_O^{R+})\}$. Figure 1 describes an iterative algorithm for computing R_O^-.

```
/* This procedure describes an algorithm for computing a
selective ruleset R_O^-. It takes an ontology O and a ruleset R as
inputs. It outputs a selective ruleset R_O^- */
PROCEDURE SELECT_RULES (O, R)
1:      r ← empty
2:      K_o ← FIND_TERMS(O)
3:      K_O^{R+} ← K_o
4:      R_O^- ← { }
5:      P_r ← { }
6:      C_r ← { }
7:      WHILE TRUE
8:          FOR ∀r ∈ R
9:              IF r ∉ R_O^- THEN
10:                 P_r ← premises (r)
11:                 IF P_r ⊆ K_O^{R+} THEN
12:                     R_O^- ← R_O^- ∪ {r}
13:                     C_r ← consequences (r)
14:                     temp ← K_O^{R+}
15:                     K_O^{R+} ← K_O^{R+} ∪ C_r
16:                 END IF
17:             END IF
18:         END FOR
19:         IF temp = K_O^{R+} THEN
20:             RETURN R_O^-
21:         END IF
22:     END WHILE
END PROCEDURE
```

Fig. 1. An algorithm for computing the selective ruleset

This procedure iteratively builds up K_O^{R+} from K_o according to already selected rules. K_O^{R+} is initialized to be K_o (line 3). For each rule r that is not already in R_O^-, it checks if P_r is a subset of K_O^{R+} (line 8-11). If yes, add r to R_O^- (line 12) and then add C_r to K_O^{R+} (line 13-15). This is because there is a chance that r will be fired adding its consequent terms into the ontology. This process iterates until a fixpoint is reached (see line 7 and line 19-20). The *premises*(*r*) and *consequences*(*r*) are two functions that correspondingly retrieve the set of premises and the set of consequents from the rule-construct dependency of *r*.

The procedure FIND_TERM in Figure 1 is used to initialize K_o. This procedure can be designed differently according to the ruleset and the set of vocabulary $\mathcal{V}$ on which the ruleset operates. Here we use a simple algorithm specifically designed for pD* rules. In general, this algorithm constructs K_o by checking the appearance of pD* constructs in O using pD* triple patterns. For example, if a match is found between a triple in O and the pD* triple pattern (*?c rdf:type rdfs:Class*), terms *rdf:type* and *rdfs:Class* will be added into K_o. All patterns in pD* rules are checked against the ontology. This algorithm is described in Figure 2. The definition for the function *match* can be found in section 2.2. To save some space, only a part of the algorithm is presented but a full list of triple patterns used in this algorithm is given in Table 1. The rest of the procedure can be inferred accordingly.

```
/* This procedure describe an algorithm for computing K_o. It takes an
ontology O as input and outputs the set of constructs used in O. */
PROCEDURE FIND_TERMS (O)
1:      K_o ← { }
2:      FOR ∀tp ∈ O
3:          IF match(tp, (?c, rdfs:domain, ?d)) THEN
4:              K_o ← K_o ∪ {rdfs:domain}
5:          END IF
6:          IF match(tp, (?c, rdf:type, rdfs:Class)) THEN
7:              K_o ← K_o ∪ {rdfs:Class, rdf:type}
8:          END IF
…       … …
        END FOR
        RETURN K_o
END PROCEDURE
```

Fig. 2. An algorithm for computing K_o based on pD* semantics

Table 1: pD* triple patterns

pD* triple patterns

?p rdfs:domain ?d	*?l rdf:type rdfs:Datatype*
?p rdfs:range ?d	*?p rdfs:subPropertyOf ?q*
?c rdfs:subClassOf ?d	*?v rdf:type ?w*
?c rdf:type rdfs:Class	*?c owl:equivalentClass ?d*
?p rdf:type rdf:Property	*?r owl:hasValue ?o*
?v owl:sameAs ?w	*?p owl:inverseOf ?q*
?r owl:allValuesFrom ?d	*?r owl:onProperty ?p*
?r owl:someValuesFrom ?d	
?p rdfs:equivalentPropertyOf ?q	
?p rdf:type rdfs:ContainerMembershipProperty	
?p rdf:type owl:FunctionalProperty	
?p rdf:type owl:InverseFunctionalProperty	
?p owl:equivalentProperty ?q	
?p rdf:type owl:TransitiveProperty	
?p rdf:type owl:SymmetricProperty	

The algorithm presented in Figure 2 should use a small amount of computational resources and hence is suitable to run on resource-constrained devices. However it is incomplete for some situations where OWL/RDFS constructs are used in uncommon ways. For example, a RDFS/OWL Full ontology could contain an assertion (*ex:subSpeciesOf rdfs:subPropertyOf rdfs:subClassOf*) where *rdfs:subClassOf* is used in the object position, which pD* triple patterns will fail to detect and therefore can lead to missing *rdfs9*, *rdfs11* and *rdfs12c* (given that *rdfs:subClassOf* is not used anywhere else in the ontology in a common way). A brute-force but effective solution could be to check each RDFS/OWL constructs used in pD* rules in all subject, predicate and object positions but this will introduce a lot more overhead.

The selective rule loading algorithm is designed to be independent of a ruleset. SELECT_RULES operates on any vocabulary $\mathcal{V}$ and ruleset R. Hence theoretically it can be applied to other OWL reasoning rulesets such as OWL 2 RL rules as well as domain-specific rules. In this paper the implementation of FIND_TERMS is designed for pD* semantics only (Figure 2). However as mentioned earlier, for each ruleset and vocabulary, FIND_TERMS can be implemented accordingly as per Figure 2. In fact, our theoretical analysis of rule-construct dependencies for OWL 2 RL rules in [43] would indicate that the selective rule loading algorithm can operate on OWL 2 RL rules. Finally, the selective rule loading algorithm is independent of reasoning algorithms, and therefore it can be applied for both rule-entailment reasoning and resolution-based reasoning.

A limitation of the selective rule loading algorithm is that it lacks flexibility in adapting the selective ruleset to an ontology with evolving terminology: the selected ruleset preserves only rules required for reasoning the original ontology, and terms added later during evolution may require deselected rules. A naïve and inefficient solution to this problem would be to re-execute selective rule loading algorithm as well as reasoning whenever a new term is added.

3.2. Two-Phase RETE Algorithm

In general the two-phase RETE algorithm applies RETE optimizations at runtime based on characteristics of a particular ontology. This algorithm is inspired by the related work in [23]. However, instead of performing a full a-priori RETE execution for collecting statistics of facts (ontology), a novel *interrupted* RETE construction mechanism is designed where statistic collection, RETE network construction and fact matching are entwined. Rather than constructing the RETE network in one go as in the original RETE, the interrupted RETE construction mechanism constructs the network in two phases, the construction of the alpha network and the construction of the beta network, and in-between a fact matching is performed on the alpha network for ontology statistics gathering. Thus the interrupted RETE construction mechanism can collect statistics required for composing a more ontology-specific and efficient RETE network without resulting in unnecessary consumption of extra resources (unlike with a full a-priori RETE execution).

The remaining part of this section explains the two-phase RETE algorithm in detail using an illustrative example comprised of an ontology snippet and two pD* rules, i.e. *rdfs9* and *rdfp15* (Figure 3). The ontology snippet basically says that *Car* is a sub-class of *Vehicle*, *Fiat* is a sub-class of *Car* (T1-T3), anything conforming to the *WithAnEngine* restriction should have a component as an *Engine* (T4-T7), *myCar* is a *Car* and one of the three components *myCar* has is an *azrTurbo* which is an *Engine* (T8-T12). By applying rule *rdfs9* and *rdfp15* to this ontology, it should infer that *myCar* is something of type *WithAnEngine* (I13), and *myCar* is also a *Vehicle* (I14). This example is crafted for illustration purposes and therefore the reasoning results may not be very interesting to practical problems. Furthermore, it contains a named restriction in order to better demonstrate the two-phase RETE algorithm, which is however rarely seen in practical OWL ontologies.

rdfs9

$$\frac{(?v\ rdfs{:}subClassOf\ ?w) \wedge (?u\ rdf{:}type\ ?v)}{(?u\ rdf{:}type\ ?w)}$$

rdfp15

$$\frac{(?v\ owl{:}someValuesFrom\ ?w) \wedge (?v\ owl{:}onProperty\ ?p) \wedge (?u\ ?p\ ?x) \wedge (?x\ rdf{:}type\ ?w)}{(?u\ rdf{:}type\ ?v)}$$

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

Fig. 3. Rule rdfs9, rdfp15 and an example ontology

3.2.1 First Phase

The first phase constructs an alpha network and gathers statistics about the ontology. To reduce memory consumption in the alpha network, an alpha node sharing mechanism is used: rule conditions with the same triple patterns share an alpha node. In this way the amount of resources required for both pattern matching and storing tokens can be shared. In the example, the condition (*?u rdf:type ?v*) from both *rdfs9* and *rdfp15* ((*?x rdf:type ?w*) in *rdfp15*) share the same alpha node.

Then, rather than continue to build a beta network as per the original RETE algorithm, the ontology is matched against the alpha network only (without the presence of a beta network). As joins are not performed, this matching should be fast. Matched triples are cached in the corresponding alpha node, and statistics about the ontology, such as the *number of matched facts per condition*, or *join selectivity factors* and so on, can then be gathered. These statistics can be used in the next phase for ontology-specific beta network construction.

In this research only the number of facts matched to each condition is collected. As will be shown in the second phase, this data can optimize join sequences in an effective manner. The (partial) RETE network for our example after the first phase should look like Figure 4.

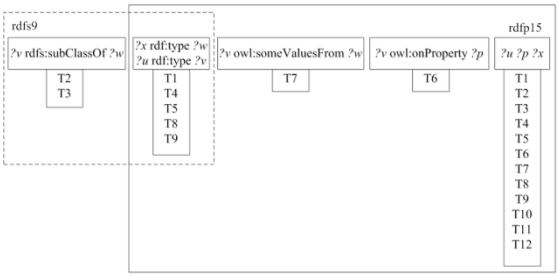


Fig. 4. The RETE network after the first phase

3.2.2 Second Phase

The second phase first builds an optimized beta network using statistics collected in the first phase, and then completes the first RETE cycle by joining facts cached in the alpha network. Two heuristics are applied to optimize the beta network: a most specific condition first heuristic and a pre-evaluation of join connectivity heuristic. Both heuristics are widely used in previous work for RETE join sequences optimization [22, 23, 32, 46, 50]. Details for the two RETE optimizations can be found in Section 2.2.

Apply Most Specific Condition First Heuristic to Optimize the First RETE Cycle

Section 2.2 introduces three criteria for evaluating a condition's specificity. Among them, using the number of facts matched to each condition when RETE reasoning ends is directly derived from the definition of condition specificity and therefore it is considered to be the most effective [32]. However to gather this data, previous research often performs a full a-priori RETE reasoning over the fact base (ontology). Furthermore, in order to find the optimal RETE network for the ontology, all possible RETE join sequences are enumerated and evaluated against complicated cost models [22, 23]. The previous approach should work well for systems requiring constant re-reasoning over the same fact base (where a one-off resource-intensive optimization is then worthwhile). However for sensor-rich systems various ontologies are often used, a-priori reasoning and evaluating all possible join structures against a complex cost model for each ontology would incur significant optimization overhead offsetting the performance/resource gains.

One may notice that statistics collected in the first phase can only represent the status of the RETE network in RETE cycle 1. As an iterative process, a RETE algorithm generates inferred facts in each RETE cycle. Hence statistics such as the number of

facts matched to each condition and join selectivity factor could vary from one RETE cycle to another. When represented using the notations introduced in Section 2.2, the number of facts matched to each condition gathered in the first phase is written as $\left|\alpha_1^{r.CDT.i}\right|: \forall r \in R \wedge 1 \leq i \leq |r.CDT|$ and in most cases $\left|\alpha_1^{r.CDT.i}\right| < \left|\alpha_l^{r.CDT.i}\right|$ where l represents the final RETE cycle.

Then the question is: will statistics collected in the first phase be sufficient for join sequence optimization? To find out the answer, an empirical study was performed which looked into the RETE networks generated for reasoning 19 small-sized ontologies. These ontologies are from various domains and with different characteristics (a list of all used ontology is given in Table 3). Results indicate that for most of studied ontologies, the majority of reasoning is carried out in the first RETE cycle: 15 out of a total of 19 ontologies have on average 75% joins performed in the first RETE cycle 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. Therefore it appears to be appropriate to order join sequences by applying optimization heuristics based on statistics collected only in the first phase. Furthermore collecting statistics in the first phase lowers the optimization overhead as required by a-priori RETE reasoning of the ontology.

Based on the discussion above, using the most specific condition first heuristic to order join sequence is then reduced to sorting conditions by the number of matched facts. Figure 5 gives the pseudo code for a simple implementation of this heuristic, based on the insertion sort algorithm. This implementation is used in our prototype resource-constrained reasoner but a better sorting algorithm could be used to enable faster condition re-ordering when a large entailment ruleset is involved. The pseudo code employs the RETE notations introduced in section 2.2. Let a $\overline{js}$ be a join sequence, the function $\overline{js}.insert$ is defined to insert a condition into the join sequence $\overline{js}$ at a specified position.

```
/* This procedure generate a join sequence for rule r using the
number of triples matched for each condition collected in the
first phase as condition specificity, i.e. α_1^{r.CDT.i}. This procedure
takes A_1^r – all alpha nodes for conditions of r after first phase–
as input and output a join sequence js optimized according to
ontology statistics.
*/
PROCEDURE ORDER_JOIN (A_1^r)
1:      js ← ( )
2:      FOR EACH α_1^{r.CDT.i} ∈ A_1^r, 1 ≤ i ≤ |r.CDT|
3:          FOR EACH j FROM 1 TO js.length
4:              IF (|js.j| ≤ |α_1^{r.CDT.i}| ≤ |js.(j + 1))
THEN
5:                  js.insert(α_1^{r.CDT.i}, j)
6:                  JUMP TO 2:
7:              END IF
8:          END FOR
9:      END FOR
10:     RETURN js
END PROCEDURE
```

Fig. 5. The RETE network after the first phase

Figure 6 shows the join sequences of our example after the application of the most specific condition first heuristics. The join sequence of *rdfs9* is already ordered with the most specific in the first and the least specific one last and hence no change is made on the join sequence. The rule *rdfp15* has the positions of the last two conditions switched, as a wildcard condition is always the least specific condition according to our chosen criterion for specificity estimation.

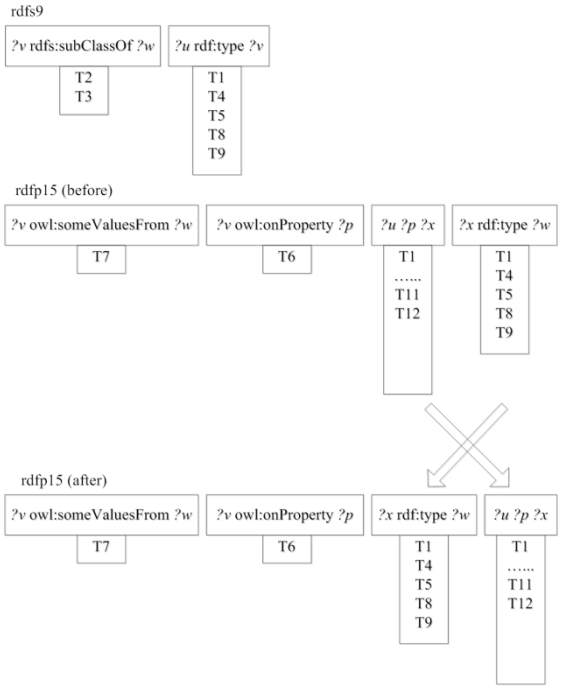


Fig. 6. An illustration of the most specific condition first heuristic in the two-phase RETE algorithm

Ensure Connectivity of an Optimized Join Sequence and Maintain As Much As Possible the Optimized Join Sequence

The *pre-evaluation of join connectivity* heuristic is applied after the most specific condition first heuristic. It ensures a join sequence is *connected*, namely a condition should share common variables with conditions it joins with. As the *pre-evaluation of join connectivity* heuristic operates on an already optimized join sequence, changing the join sequence for connectivity will to some extent impair the previous optimization. Therefore the aim here is to ensure the connectivity of a join sequence in the meantime maintains as much as possible of the already ordered join sequence.

Similarly, let $\overline{js}$ be the join sequence optimized by the most specific condition first heuristic, $\overline{js}.i: (1 \leq i \leq |\overline{js}|)$ the ith condition, $\overline{js}_{1\rightarrow i}: (1 < i \leq |\overline{js}|)$ the partial join sequence from $\overline{js}.1$ to $\overline{js}.i$. Our algorithm for the pre-evaluation of join connectivity heuristic then scans $\overline{js}$ from start to end. For each condition $\overline{js}.n$ that is found not connected to $\overline{js}_{1\rightarrow n-1}$, the algorithm inserts in front of $\overline{js}.n$ the *first* condition $\overline{js}.m: n < m \leq |\overline{js}|$ that is connected to $\overline{js}_{1\rightarrow n-1}$. Therefore $\overline{js}.m$ is the most specific condition in the join sequence after $\overline{js}.n$ that can connect to $\overline{js}_{1\rightarrow n-1}$. If there is no such a $\overline{js}.m$, then $\overline{js}.n$ is left in place, and the algorithm continues to check the connectivity of $\overline{js}.(n+1)$ in the same way. Figure 7 describes the above algorithm in pseudo code.

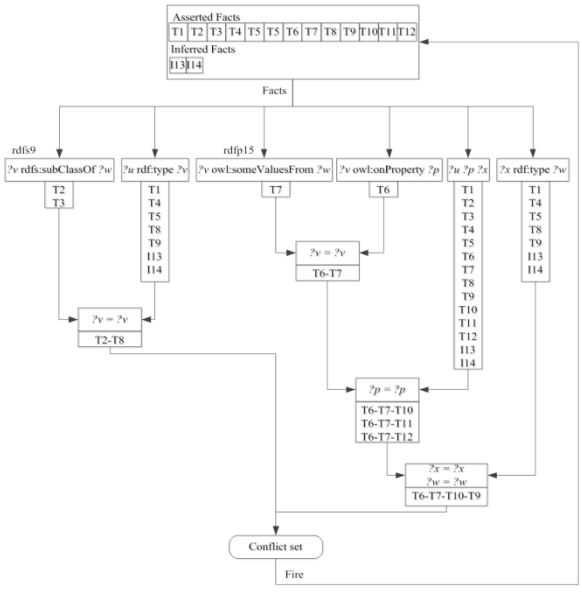
```
/* This procedure implements an algorithm for checking join
connectivity. It takes a join sequence js as input and output a
js. This procedure
 */
PROCEDURE CONNECT_CONDITIONS (js)
1:      var_prev ← { }
2:      var ← { }
3:      FOR j FROM 2 TO js.length
4:          var_prev ← variables(∀js.i, 1 ≤ i ≤ j − 1)
5:          var ← variables(js.j)
6:          IF var_prev ∩ var = { } THEN
7:              ins ← empty
8:              FOR m FROM j TO js.length
9:                  IF variables(js.m) ∩ var_prev ≠ { }
THEN
10:                     ins ← js.m
11:                 END IF
12:             END FOR
13:             IF ins ≠ empty THEN
14:                 js.remove(ins)
15:                 js.insert(ins, j)
16:             ELSE
17:                 j ← j+1
18:                 JUMP TO 4:
19:             END IF
20:         END IF
21:     END FOR
22:     RETURN js
END PROCEDURE
```

Fig. 7. The RETE network after the first phase

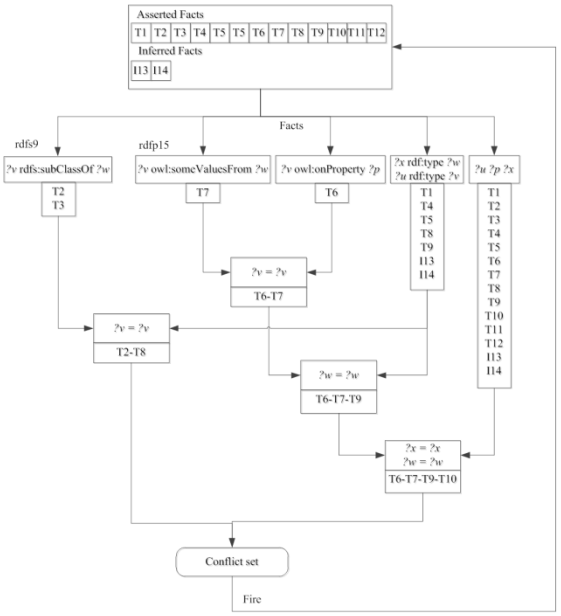
The function *variables* receives conditions as arguments and returns the set of variables used in given conditions. The function $\overline{js}.remove$ removes a specified condition from the join sequence. Line 6 and 9 respectively check the connectivity between $\overline{js}_{1\rightarrow j-1}$ and $\overline{js}.j$ and between $\overline{js}_{1\rightarrow j-1}$ and $\overline{js}.m$. As the join sequences for *rdfs9* and *rdfp15* in our example are already connected, the application of this heuristic will not change their join sequences.

Build a Beta Network According to the Optimized Join Sequences and Finish the RETE Reasoning

The two-phase RETE algorithm then constructs a beta network according to the join sequence produced by CONNECT_CONDITIONS. The first RETE cycle resumes and tokens stored in the alpha memory (because of the initial matching for statistics collection) are passed along the beta network joining to each other as per the normal RETE algorithm. Figure 8 compares the two example RETE networks after all facts are matched. Tokens are listed in the box under the corresponding alpha/beta nodes. It can be seen that the amount of tokens in the alpha network of Figure 8b is smaller than those in the counterpart alpha network in Figure 8a, because of the use of the node sharing mechanism. The re-ordering of the join sequence of rule rdfp15 causes a reduction of cached tokens in the beta network. Both algorithms correctly deduce tokens I13 and I14.



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



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

Fig. 8. The complete RETE network

3.3.3 Discussion

It is clear that the two-phase RETE algorithm is only applicable to RETE-based algorithms, but selective rule loading algorithm is independent of any rule-based reasoning algorithms, as long as RS rules are used. As pointed out earlier, the selective rule loading algorithm will need re-execution to handle a changing ontology. However unlike the selective rule loading algorithm, evolving terminology is not a problem for the two-phase RETE algorithm as it loads the entire ruleset. Thus changes of the ontology can be reflected immediately in the reasoning without re-executing the entire composition.

Only the number of facts matched to each condition in the first phase is used here to optimize join sequences. This could lead to a situation that join sequences optimized according to this statistic are effective mostly for the first (few) RETE cycle (although through our study it has been observed that most operations and resource consumption happen in the first RETE cycle). We envisage that with more diverse types of ontology statistics, optimized join sequences can be even more effective not only for the first (few) RETE cycles but for subsequent RETE cycles. It is also important to note that the two-phase RETE algorithm optimisation is

orthogonal to the selective rule-loading algorithm. As a result the authors would argue that with some code-level modification, none, one, or both of the two reasoner optimisations could be applied in any rule-entailment reasoner implementation.

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. COROR is only implemented to investigate if the reasoner composition approaches designed can reduce resource consumption for rule-entailment reasoning. Therefore the minimal number of components was implemented with core components being the composable RETE engine and a framework for loading, parsing and manipulating ontologies. Many other components of practical use, such as a query interface, a remote reasoning interface and so on, were omitted from this minimal implementation. However they should work well with our composition algorithms. An overview of the COROR components is given in Figure 9.

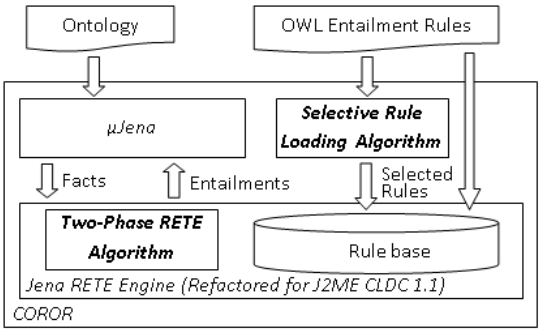


Fig. 9. An overview of COROR

The target platform for COROR is Sun SPOT sensor platform v4.0 (blue)[4]. Hardware and software specifications of this platform are given in Table 2. 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 supporting Java programming using Netbeans and testing using a fully functional emulator.

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

Table 2. 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 COROR component from scratch, it was decided to refactor and reuse some existing software. The µJena framework [49] is a cut-down version of Jena for J2ME. It is used in COROR for ontology parsing and manipulation. However as reasoning capability has been removed from µJena, the Jena RETE engine was refactored and ported to run on µJena.

Triple is the only type of fact for Jena RETE engine (and therefore COROR). A triple in COROR is comprised of three *RDF nodes* each of which respectively represents its subject, predicate or object. RDF nodes used in the ontology are maintained in a RDF *node pool* in COROR. Only one real node is constructed for a URI and the reference to the RDF node is used in different triples. Types of RDF nodes that can be used in a triple include *URI nodes*, *literal nodes*, and *blank nodes*. Besides the above three RDF node types, a triple pattern can also have *rule variable nodes* to represent variables. A URI node and a literal node keep the actual URI/literal string in the node. All triples are stored in a *triple pool* implemented using a J2ME Vector object. No index is built for the triple pool nor for all alpha/beta memories. Hence accessing the triple/token is performed using linear search. The triple pool also serves as the working memory for the RETE engine. Tokens used in the RETE network were implemented to use a dual array: one array for rule variables and the other for bound values of corresponding variables. The reason for the dual array is to enable node sharing.

To improve resource efficiency for triple storage, a hash of URI/literal strings or an indexed mapping dictionary between identifiers (e.g. a Long number) and URI/literal strings can speed up node searching and comparison. It is worth noting that as our research concentrates on the amount of resources that can be reduced by reasoner composition algorithms, very few software engineering level optimizations were used. However, such optimizations should not be exclusively applicable to our composition algorithms and they can be implemented on a more stable COROR implementation.

The pD* semantics was chosen to be the semantics for COROR. It was implemented as 39 entailment rules in the Jena format: including 16 D* entailment rules and 23 P entailment rules. Rules for detecting clashes (XML-clash, part D-clash, P-clash, see more about clashes in [44]) are not handled in this implementation. This is consistent with the assumption made by 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].

The choice of pD* rules instead of OWL 2 RL rules is motivated by a number of reasons. First, at the time the implementation and the evaluation were undertaken there was not a set of Jena compatible OWL 2 RL rules available. Second, compared to pD* rules implementing OWL 2 RL rules as Jena rules could be more error-prone (given that RDF list is heavily involved). Third, as opposed to constructing a fully-fledged resource-constrained reasoner, the main aim of this work is to study how reasoner composition can reduce resource consumption for rule-entailment reasoning. Hence pD* was chosen as a more straightforward way to carry out this research. Nevertheless, the composition algorithms introduced in Section 3 are independent of rulesets and therefore using pD* rules does not exclude the possibility of OWL 2 RL rules in the future. In section 3.5 of [43] rule-construct dependencies of OWL 2 RL rules are discussed. It establishes that OWL 2 RL can work with composition algorithms. Considering the shift from OWL 1 to OWL 2 for most Semantic Web tools in the last few years, this topic will be studied in the near future.

The analysis of rule-construct dependencies for pD* rules was performed manually. It was a once-off effort and rule-construct dependencies are kept as plain text in a local text file. The rule-construct dependencies file is loaded and parsed in memory at the beginning of selective rule loading algorithm. In the future when domain-specific rules are also integrated, an automatic rule analysis procedure needs to be developed.

Composition algorithms are implemented to work together and individually. We use different names to distinguish the combinations of composition algorithms. They are 1: *COROR-noncomposable*, 2: *COROR-selective*, 3: *COROR-two-phase*, and 4: *COROR-hybrid*. The above modes respectively correspond to the use of none, one, or two composition algorithms. In the hybrid mode, selective rule loading algorithm first produces a selective ruleset and then the two-phase RETE algorithm builds a customized RETE network according to the ontology to be reasoned over.

5. Evaluation

The aim of this evaluation was to explore if, and to what extent reasoner composition can reduce resource consumption for rule-entailment OWL reasoning. To do this, the evaluation tested and compared the performance of the different COROR composition modes to reason over ontologies. Two items need to be clarified here: (1) the term reasoning refers to computing all entailments of ontology according to the given ruleset, and (2) the term performance refers to both time performance and memory performance.

5.1. Experiment Design and Execution

Two experiments were performed. The first experiment was an *intra-reasoner comparison* that investigated the change of the reasoning performance of COROR with and without our composition algorithms. In our case, four COROR composition modes were compared side by side. The second experiment was an *inter-reasoner comparison* that compares our composable rule-entailment reasoner with some other rule-entailment reasoners from the state of the art. This second experiment studied if a composable rule-entailment reasoner would be more suitable to run on a resource-constrained device in comparison with off-the-shelf rule-entailment reasoners (from a performance perspective). To do this second experiment, COROR-hybrid was compared with 4 other in-memory rule-entailment reasoners: Bossam 0.9b45, Jena v2.6.3 (with forward reasoning only, termed Jena-forward in this paper), BaseVISorv1.2.1, and swiftOWLIM v3.0.10. COROR-hybrid was used because it represents the performance gain of both modes together. Selected reasoners have similar semantics to that of COROR, avoiding the bias that one reasoner is slower or 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 ruleset as COROR; the expressivity of Bossam (closed source) was not

defined at the time on either its website [5], publications [24], or implementation, which made it difficult to judge its inference capability. Pellet is also included only to give readers an impression of how COROR performs compared to a full-fledged, complete DL-tableau reasoner. However since the expressivity of Pellet is larger than pD*, it is unfair to compare the performance of Pellet with COROR side by side, and this is also not our intention.

Memory and time used by each reasoner to reason over an ontology are measured for the performance comparison. Reasoning time is measured by taking the difference of the time recorded before and after reasoning, using *System.CurrentTimeMills*(). Memory required by reasoning is measured by subtracting the free memory from the total memory (*Runtime.getRuntime().totalMemory()-Runtime.getRuntime().freeMemory()*) after the reasoning, when the RETE network reaches its maximum size (all entailments are fully calculated and all tokens are cached) and memory measured at this moment should reach its maximum.

Measurements for time and memory were performed separately in different executions to reduce interference between each other. The Java methods used to measure memory are not particularly reliable. They produce only an approximation of the memory usage. To reduce the statistical errors in each time/memory measurement, each result of time/memory used in the evaluation is an average of 10 individual measurements. Furthermore garbage collection was performed to release garbage so interference from non-recycled garbage memory could be reduced as much as possible. This was done by explicitly calling *System.gc()* 20 times before each memory measurement. Therefore the results should be sufficiently accurate for the evaluation goal of this paper. A threshold of 30 minutes was set to avoid excessively long reasoning. Any reasoning process still running longer than this threshold 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. The inter-reasoner comparison was performed on a desktop computer as reasoners selected for comparison were written in J2SE and cannot run on Sun SPOT (which runs J2ME CLDC 1.1). It is non-trivial to port them to a J2ME CLDC 1.1 platform, but COROR can run natively on a J2SE platform. The experiment environment for inter-reasoner comparison was: Eclipse Helios with Java SE 6 Update 14 with 128MB maximum heap size, Dual Core CPU @ 2.4GHz, 3.25GB RAM.

Commonly used ontology benchmarking tools such as LUBM [66] could have been used to carry out performance experiments but even the smallest dataset generated by LUBM (synthetic data about one department in an university) contains from 6000 to 8000 triples, which requires more than the threshold for COROR to reason on a Sun SPOT. For our evaluation we selected more appropriate smaller ontologies (as listed in Table 3). The expressivity, number of classes/properties/individuals, and the size both before and after COROR reasoning are listed (in number of triples). The selection of the ontologies was based on three factors: (1) they are from different domains, which to some extent can represent the diversity of characteristics of ontology used in resource-constrained systems, (2) they vary in expressivities, 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 3: Characteristics of ontologies used in the experiments

	Expressivity	Cls	Prop	Indv	Before	After
teams	$\mathcal{ALCIN}$	9	3	3	87	497
owls-profile	$\mathcal{ALCHIOF}$(D)	54	68	13	116	594
Koala	$\mathcal{ALCON}$(D)	20	7	6	147	710
university	$\mathcal{SIOF}$(D)	30	12	4	169	760
Beer	$\mathcal{ALHI}$(D)	51	15	9	173	933
mindswapper	$\mathcal{ALCHIF}$(D)	49	73	126	437	1350
Foaf	$\mathcal{ALCHIF}$(D)	17	69	0	503	1772
mad_cows	$\mathcal{ALCHOIN}$(D)	54	17	13	521	1695
Biopax	$\mathcal{ALCHF}$(D)	28	50	0	633	2228
Food	$\mathcal{ALCOF}$	65	10	57	924	2437
mini-tambis	$\mathcal{ALCN}$	183	44	0	1080	3407
amino-acid	$\mathcal{SHOF}$(D)	55	24	1	1465	2932
atk-portal	$\mathcal{ALCHIOF}$(D)	169	147	75	1499	5418
Wine	$\mathcal{SHOIN}$(D)	77	16	161	1833	6376
Pizza	$\mathcal{ALCF}$(D)	87	30	0	1867	3819
tambis-full	$\mathcal{SHIN}$	395	100	0	3884	10959
Nato	$\mathcal{ALCF}$(D)	194	885	0	5924	15746

5.2. Results and Discussions

This section presents and discusses results of the experiments undertaken.

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

5.2.1 Intra-reasoner comparison

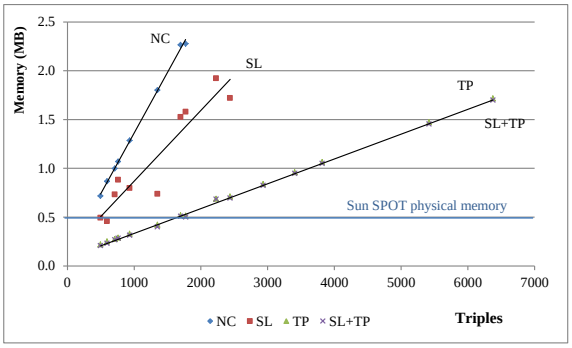
Time and memory consumptions required by the four COROR composition modes in the intra-reasoner comparison are listed in Table 4. For an ontology that requires time more than the preset threshold (30 minutes), manual termination is imposed and therefore no results were collected for these cases.

Table 4: Results for inter-reasoner comparison (time in Second and memory in KB). NC = COROR-noncomposable; SL = COROR-selective; TP = COROR-Two-Phase, SL+TP = COROR-hybrid

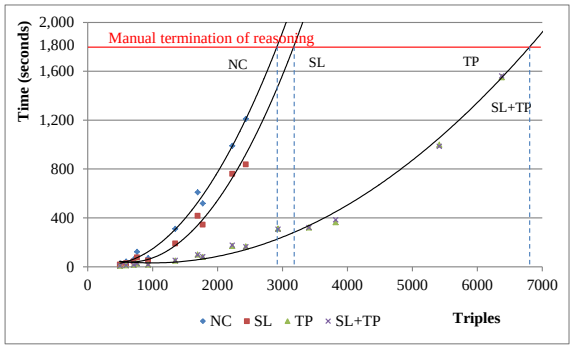
	NC		SL		TP		SL+TP	
	Time	**Mem**	**Time**	**Mem**	**Time**	**Mem**	**Time**	**Mem**
teams	24.7	717	17.4	493	7.5	219	7.5	208
profile	41.9	867	27.2	459	11.5	251	11.6	232
koala	58.7	997	42.3	732	14.8	276	15.0	268
beer	70.1	1284	48.9	797	16.8	329	17.3	313
university	121.3	1069	76.0	883	25.8	289	25.9	284
mindswappers	309.5	1800	189.4	738	49.5	422	50.3	401
foaf	517.6	2276	343.2	1579	80.6	514	81.8	500
mad_cows	609.2	2267	416.1	1526	98.1	518	97.5	509
biopax-level1	N/A	N/A	760.2	1922	169.8	688	176.3	683
food	N/A	N/A	836.3	1721	164.8	711	163.7	696
miniTambis	N/A	N/A	N/A	N/A	321.3	960	320.1	947
amino-acid	N/A	N/A	N/A	N/A	307.3	838	307.3	824
atk-portal	N/A	N/A	N/A	N/A	999.2	1472	984.7	1455
wine	N/A	N/A	N/A	N/A	1550.6	1716	1561.2	1698
pizza	N/A	N/A	N/A	N/A	364.3	1064	384.3	1049
tambis-full	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
nato	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

As shown in Table 4, a composable COROR (COROR with one or two composition algorithms enabled, refer to columns SL, TP, SL+TP) greatly outperforms a noncomposable COROR (COROR with no composition algorithm enabled, refer to column NC) in both memory and time. COROR-selective uses on average 35% less memory than COROR-noncomposable (with a std. dev. 13%) and on average 33% less time (with a std. dev. 4%). COROR-hybrid uses on average 74% less memory than COROR-noncomposable (with a std. dev. 3%) and on average 78% less time (with a std. dev. 6%). COROR-hybrid uses on average 75% less memory than COROR-noncomposable (with a std. dev. 3%) and on average 78% less time (with a std. dev. 6%). For 8 ontology (i.e. *teams*, *owls-profile*, *koala*, *university*, *beer*, *mindswappers*, *mad_cows*, and *foaf*) COROR-hybrid requires less than the RAM size of Sun SPOT (512KB) to perform reasoning, and this number is 6 for COROR-two-phase, 2 for COROR-selective, but none for COROR-noncomposable. Whenever the reasoning process cannot be achieved only in RAM, extensive paging of RAM state to/from (Flash) storage is required, delaying reasoning time and costing more power consumption.

Figure 10a and Figure 10b show the correlation between memory/time consumption and the sizes of reasoned ontology. The choice of sizes of reasoned ontology rather than sizes of original ontology is because memory measurements are collected at the end of reasoning when all inferences are deduced. Trend lines are plotted for different COROR modes. In general the memory consumption shows a linear increase with size of reasoned ontology. The different slopes of trend lines indicate that the memory consumption of COROR-noncomposable increases much faster than the other COROR modes. Reasoning time shows a quadratic polynomial increase with size of reasoned ontology. Similarly, COROR-noncomposable shows a much steeper increase in time than the rest. The time required by COROR-selective to perform reasoning is less than COROR-noncomposable but still shows a much steeper increasing trend than that of the other two composable COROR modes.



(a) Memory vs No. triple after reasoning



(b) Time vs No. triple after reasoning

Fig. 10. Time/Memory vs No. triple after reasoning

Figure 10b also enables the estimation of the maximum ontology size that different COROR modes can handle under evaluation settings (Sun SPOT, 30 minutes time threshold). From Figure 10b, it can be seen from the trend lines that: with a 30 min threshold, 180MHz CPU and 512KB RAM, the maximum supported ontology size is around 2900 triples (after reasoning) for COROR-noncomposable, around 3200 triples (after reasoning) for COROR-selective, around 6800 triples (after reasoning) for COROR-two-phase and COROR-hybrid. Knowing the maximum ontology size supported by each COROR mode could allow users to determine whether an ontology is suited for reasoning with COROR with given hardware specifications and threshold requirement. This can be achieved by first reasoning over the ontology using a desktop rule-entailment reasoner using pD* rules, and then collecting the size of the reasoned ontology. By adjusting the time threshold, COROR will have a different maximum supported ontology size.

Although both COROR-two-phase and COROR-selective greatly reduce the reasoning time and memory consumption, it was observed that in the experiment COROR-hybrid performed no better than COROR-two-phase. To investigate this observation, an in-depth examination was performed of RETE networks constructed by the four COROR modes. In general match operations and join operations dominate the execution of a RETE algorithm. We therefore chose the number of match operations (#M) and the number 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 tokens generated and cached in the network (#TK) was selected as an indicator of memory usage for COROR, and hence the amount of tokens generated and cached in alpha network ($\#TK_\alpha$) and beta network ($\#TK_\beta$) was therefore the respective indicator for memory consumption of alpha network the beta network.

By measuring these indicators, it was noticed that compared with COROR-noncomposable, COROR-two-phase has less #M and $\#TK_\alpha$ but has almost the same #J and $\#TK_\beta$. This implies that the memory and time gain of COROR-selective came mainly from alpha nodes of deselected rules but the beta network benefit little from deselected rules. A close examination of the deselected rules shows that their join sequences are already optimized by the original pD* rule author. Hence in an execution of a COROR-noncomposable reasoning, even when these deselected rules are loaded into the RETE engine, they still perform very few superfluous join operations, leading to a nearly empty beta network. For this reason, deselecting these rules in executions of COROR-selective does 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 but Table 4 indicates that COROR-two-phase yielded much more performance gain than that of COROR-selective. Examination of the RETE networks generated by COROR-two-phase executions reveal 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 $\#TK_\beta$. In 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 $\#TK_\beta$. For rdfp15, the last two conditions in the join sequence are switch over according to the *most specific condition first* heuristic, leading to a small reduction of #J and $\#TK_\beta$. However, as the rule rdfp15 accounts for a very small portion of the total #TK in the whole RETE network (only 0.91% for the *wine* ontology and even smaller for the other tested ontology), the performance gain from reordering the join sequence of rdfp15 is subtle.

From the above analysis, we can deduce that *the reason that COROR-selective achieves less performance improvement than that of COROR-two-phase is mainly 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/#TK_β per rule and hence the more performance gain from a shared alpha network. On the other hand, it also shows that for rulesets with highly shared conditions a shared alpha network can save more resources in alpha network than merely removing parts of the alpha network. Since COROR-hybrid is only a simple combination of the selective rule loading algorithm and the two-phase RETE algorithm, the above analysis also explains the very similar performance of COROR-two-phase and COROR-hybrid (Table 4).

To establish that the heuristics introduced in the second phase of the two-phase RETE algorithm do lead to a 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 Table 4 (on the same Sun SPOT emulator with the same set of ontologies), using both (original and modified) rulesets. 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 ruleset is used.

rdfp1

(*?p rdf:type owl:FunctionalProperty*) ∧ (*?u ?p ?v*) ∧ (*?u ?p ?w*) ∧ notLiteral(*?v*)

(*?v owl:sameAs ?w*)

rdfp1m

(*?u ?p ?v*) ∧ (*?u ?p ?w*) ∧ (*?p rdf:type owl:FunctionalProperty*) ∧ notLiteral(?v)

(*?v owl:sameAs ?w*)

rdfp2

(*?p rdf:type owl:InverseFunctionalProperty*) ∧ (*?u ?p ?w*) ∧ (*?v ?p ?w*)

(*?u owl:sameAs ?v*)

rdfp2m

(*?u ?p ?w*) ∧ (*?v ?p ?w*) ∧ (*?p rdf:type owl:InverseFunctionalProperty*)

(*?u owl:sameAs ?v*)

rdfp4

(*?p rdf:type owl:TransitiveProperty*) ∧ (*?u ?p ?v*) ∧ (*?v ?p ?w*)

(*?u ?p ?w*)

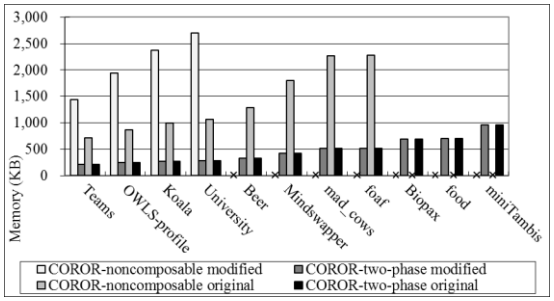
rdfp4m

(*?u ?p ?v*) ∧ (*?v ?p ?w*) ∧ (*?p rdf:type owl:TransitiveProperty*)

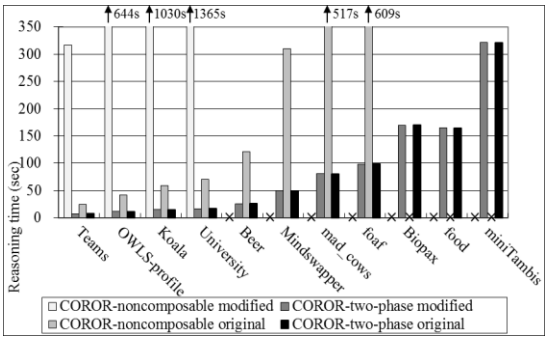
(*?u ?p ?w*)

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

It is apparent from Figure 12a and Figure 12b that COROR-noncomposable-modified uses a lot more memory and time than COROR-noncomposable-original does in reasoning over the same ontology. Compared with results presented in Table 4, 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 that poor performance will result in running a set of less optimized rules in a noncomposable reasoner. However, the use of the modified rules does not change much in time/memory performance for COROR-two-phase. Inspection into the RETE networks showed that join sequences of modified rules are reordered by join sequence reordering heuristics in two-phase RETE algorithm to the same join sequences as they are in the original ruleset. 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 rulesets manually reordered by rule-authoring specialists, with the added advantage that the reordering applied in COROR is also tuned to the particular ontology, and procuring a hand-optimised ruleset for each ontology is infeasible. Sub-optimal join ordering would also be common in manually authored application-specific rules since these rules would be written by domain experts rather than rule-authoring-experts. Moreover, compared to the reduction of #M and #TK_α in COROR-two-phase-modified, the reduction of #J and #TK_β in COROR-two-phase-modified is more substantial, therefore indicating that the majority performance gain comes from the join sequence reordering in the second phase of the two-phase RETE algorithm.



(a) Memory performance



(b) Time performance

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

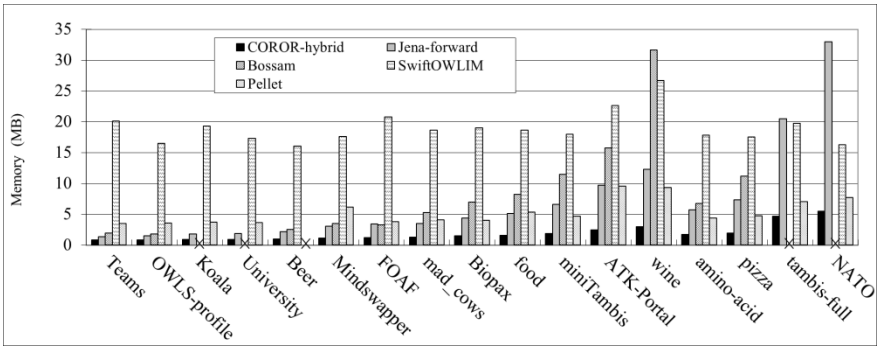
Based on the above discussion, it can be inferred that the two-phase RETE algorithm would be better at handling a ruleset with long and sub-optimal join sequences and with highly shared conditions, whilst the selective rule loading algorithm would work better on rules with shorter join sequences (no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

5.2.2 Inter-reasoner comparison

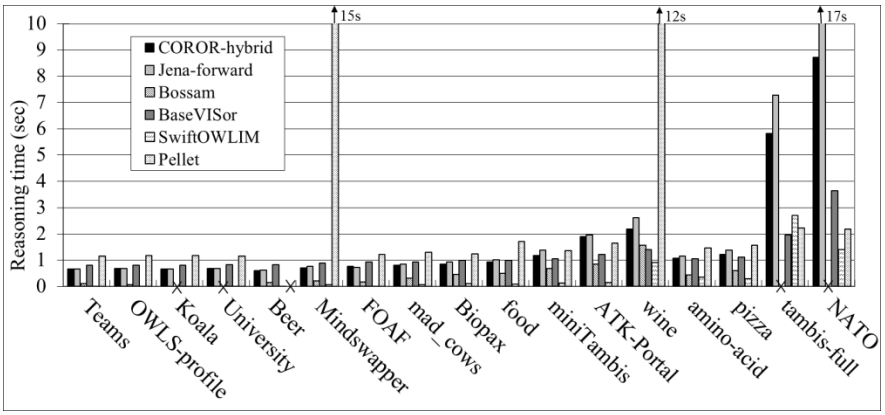
Results of the inter-reasoner comparison are given in Figure 13 (memory performance in Figure 13a and time performance in Figure 13b). As shown in the figures, the time performance of COROR-hybrid is comparable to Jena-forward and BaseVISor. However, in contrast to results of the intra-reasoner comparison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

SwiftOWLIM is 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 some ontologies. For smaller ontologies such as *Teams*, *OWLS-profile*, *Beer*, Bossam can compete with SwiftOWLIM, but it fails with errors for four ontology including *Koala*, *University, tambis-full and NATO* all of which are successfully reasoned over by other reasoners. Pellet generally has quite constant performance for most selected ontologies regardless of their sizes. However for smaller ontologies it uses more time than the other rule-entailment reasoners. Pellet requires much more time to reason over *wine* and *mindswapper*. This is because very expressive structures used in the terminology (which in the Wine ontology were specifically designed to stress the reasoner) slow down the DL tableau reasoning process. Pellet does not have results for the *Beer* ontology because of inconsistencies.

In the experiment the memory performance of COROR is much better than all other reasoners for the tested ontologies. This is especially the case for smaller ontologies. The memory usage of Bossam and Jena-forward grows much faster than that of COROR-hybrid as the size and the complexity of an ontology increases. For example, for *ATK-portal* Bossam uses 6 times more memory than COROR and for *Wine* it uses 13 times more memory than COROR. The memory footprint for swiftOWLIM is much larger than COROR even for very small ontology. It uses 20MB of memory to reason over *teams* which is 15 times larger than that used by COROR. This shows that swiftOWLIM trades increased memory usage for time improvements. BaseVISor hides its reasoning process from external inspection so it is not possible to measure its memory usage and therefore it is missing from the memory comparison. The overall results indicate that a much smaller memory footprint is needed for COROR (-hybrid) to reason over small sized ontologies without sacrificing time performance. This demonstrates that from a performance perspective it is much more feasible to use COROR in 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, and so on, and all reasons 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. Our 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, were designed to compose an optimized reasoning process, without sacrificing the generality or semantics supported in the reasoner. A performance evaluation of a naïve proof-of-concept implementation of the algorithms indicates that reasoner composition based only on optimising the ruleset used can save on average 35% of the reasoning memory and 33% of the reasoning time, and composition based on extending 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 partially 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 and Science Foundation Ireland via grant 08/SRC/I1403 ("Federated, Autonomic Management of End-to-End Communications Services").

Reference

[1] A. Acciarri, D. Calvanese, G. D. Giacomo, D. Lembo, M. Lenzerini, M. Palmieri, R. Rosati, "QuOnto: querying ontologies". In M. Veloso and S. Kambhampati, Eds., Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, 2005, pp. 1670-1671, AAAI Press / The MIT Press.

[2] W. Tai, J. Keeney, and D. O'Sullivan, "COROR: A COmposable Rule-entailment Owl Reasoner for Resource-Constrained Devices". In N. Bassiliades, G. Governatori and A. Paschke, Eds., Proceedings of The 5th International Symposium on Rules: Research Based and Industry Focused (RuleML'11), 2011, pp. 212–226, Springer.

[3] S. Ali and S. Kiefer, "μOR – A Micro OWL DL Reasoner for Ambient Intelligent Devices". In N. Abdennadher and D. Petcu, Eds., Proceedings of the 4th International Conference on Advances in Grid and Pervasive Computing, 2009, pp. 305–316, Springer.

[4] P. Wang, J. Zheng, L. Fu, E. W. Patton, T. Lebo, L. Ding, Q. Liu, J. Luciano, and D. McGuinness, "A semantic portal for next generation monitoring systems". In L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. Fridman Noy and E. Blomqvist, Eds., Proceedings of the 10th International Semantic Web Conference (ISWC'11), pp. 253–268, 2011, Springer.

[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] G. Cheng, S. Gong, and Y. Qu, "An Empirical Study of Vocabulary Relatedness and Its Application to Recommender Systems". In L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. Fridman Noy and E. Blomqvist, Eds., Proceedings of the 10th International Conference on The Semantic Web (ISWC2011), 2011, pp. 98–113, Springer.

[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 H. de Swart, Eds., Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX '98, 1998, pp. 60-76, Springer.

[9] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov, "OWLIM: A family of scalable semantic repositories" Semantic Web, 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 Philipp. Cimiano and H. S. Pinto, Proceedings of the International Conference on Knowledge Engineering and Management by the Masses, EKAW, 2010, pp. 565–575, Springer.

[11] Y. Kazakov, M. Krötzsch, and F. Simancík, "Concurrent Classification of EL Ontologies". In L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. Fridman Noy and E. Blomqvist, Eds., Proceedings of the 10th International Semantic Web Conference (ISWC'11), 2011, pp. 305–320, Springer.

[12] J. J. Carroll, D. Reynolds, I. Dickinson, A. Seaborne, C. Dollin, and K. Wilkinson, "Jena: Implementing the Semantic Web Recommendations". In S. I. Feldman, M. Uretsky, M. Najork and C. E. Wills, Eds., Proceedings of the 13th International World Wide Web Conference, 2004, pp. 74-83, ACM.

[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/. Last accessed 1-July-2014.

[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 P. Pichappan and A. Abraham, Eds., Proceedings of International Conference on Digital Information Management, 2008, pp. 172-177, IEEE.

[15] P. Desai, C. Henson, P. Anatharam, and A. Sheth, "Demonstration : SECURE -- Semantics Empowered ResCUe Environment". In K. Taylor, A. Ayyagari and D. De Roure Proceedings of the 4th International Workshop on Semantic Sensor Network, 2011, pp. 115–118, CEUR-WS.org.

[16] U. Hustadt, B. Motik, and U. Sattler, "Reducing SHIQ-description logic to disjunctive datalog programs". In D. Dubois, C. A. Welty and M. Williams, Eds., Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004), 2004, pp. 152–162, AAAI Press.

[17] "SPIN –SPARQL Inference Notation" 2012. [Online]. Available: http://spinrdf.org/. Last accessed 1-July-2014.

[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 G. Hencsey, B. White, Y. R. Chen, L. Kovács and S. Lawrence, Eds., 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/. Last accessed 1-July-2014.

[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 H. E. Shrobe, T. M. Mitchell and R. G. Smith, Eds., Proceedings of the 7th National Conference on Artificial Intelligence (AAAI'87), 1988, pp. 699-704, AAAI Press / The MIT Press.

[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 G. Antoniou and H. Boley, Eds., Proceedings of International Workshop on Rules and

Rule Markup Languages for the Semantic Web, 2004, pp. 128-138, Springer.
[25] "Hoolet reasoner" 2012. [Online]. Available: http://owl.man.ac.uk/hoolet/. Last accessed 1-July-2014.
[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, IEEE Computer Society.
[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 K. Taylor, A. Ayyagari and D. De Roure, Eds., Proceedings of the 4th International Workshop on Semantic Sensor Network, 2011, pp. 69–81, CEUR-WS.org.
[28] C. Matheus, K. Baclawski, and M. Kokar, "BaseVISor: A Triples-Based Inference Engine Outfitted to Process RuleML and R-Entailment Rules". In T. Eiter, E. Franconi, R. Hodgson and S. Stephens, Eds., Proceedings of International Conference on Rules and Rule Markup Languages for the Semantic Web, pp. 67-74, Nov. 2006, IEEE Computer Society.
[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". In B. Grau, I. Horrocks, B. Motik and U. Sattler, Eds. Proceedings of the 2009 International Workshop on Description Logics, 2009.
[31] "The Gene Ontology" 2012. [Online]. Available: http://www.geneontology.org/. Last accessed 1-July-2014.
[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] A. Hogan, J. Umbrich, A. Harth, R. Cyganiak, A. Polleres, and S. Decker, "An empirical survey of Linked Data conformance", Web Semantics: Science, Services and Agents on the World Wide Web, vol. 14, pp. 14–44, Jul. 2012.
[34] C. Seitz and R. Schönfelder, "Rule-based OWL reasoning for specific embedded devices". In L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. Fridman Noy and E. Blomqvist, Eds., Proceedings of the 10th International Semantic Web Conference (ISWC'11), 2011, pp. 237–252, Springer.
[35] T. Kawamura and A. Ohsuga, "Toward an ecosystem of LOD in the field: LOD content generation and its consuming service". In P. Cudré-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Euzenat, J.X. Parreira, J. Hendler, G. Schreiber, A. Bernstein and E. Blomqvist, Eds., The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part II, 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 Robert Nieuwenhuism, Eds., Proceedings of International Conference on Automated Deduction, 2005, pp. 452-457, Springer.
[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 C. Dolbear, A. Ruttenberg and U. Sattler, Eds., Proceedings of the 5th International Workshop on OWL: Experiences and Directions, 2008, vol. 432, CEUR-WS.org.
[40] V. Vassiliadis, J. Wielemaker, and C. Mungall, "Processing OWL2 ontologies using Thea : An application of logic programming". In R. Hoekstra and P. F. Patel-Schneider, Eds., Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2009), 2009, CEUR-WS.org.
[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. Last accessed 1-July-2014.
[42] "Witmate, the only one pure java Logic/Rule Engine executable from J2ME to J2SE/J2EE." [Online]. Available: http://www.witmate.com/default.html. Last accessed 1-July-2014.
[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 U. Furbach and N. Shankar, Eds., Proceedings of the International Joint Conference on Automated Reasoning (IJCAR 2006), 2006, pp. 292-297, Springer.
[46] Y.-wang Wang and E. N. Hanson, "A Performance Comparison of the Rete and TREAT Algorithms for Testing Database Rule Conditions". In F. Golshani, Eds., Proceedings of the 8th International Conference on Data Engineering (ICDE'92), 1992, pp. 88-97, IEEE Computer Society.
[47] Y. Zou and T. Finin, "F-OWL: An inference engine for semantic web". In M. G. Hinchey, J. L. Rash, W. Truszkowski and C. Rouff, Eds., Workshop on Formal Approaches to Agent-Based Systems, 2004, pp. 238-248, Springer.
[48] J. Zhou, L. Ma, Q. Liu, L. Zhang, Y. Yu, and Y. Pan, "Minerva: A Scalable OWL Ontology Storage and Inference System". In R. Mizoguchi, Z. Shi and F. Giunchiglia, Eds., Proceedings of the 2006 Asian Semantic Web Conference (ASWC06), 2006, pp. 429-433, Springer.
[49] F. Crivellaro, G. Genovese, and G. Orsi, "µJena - Software." [Online]. Available: http://poseidon.ws.dei.polimi.it/ca/?page_id=59. Last accessed 1-July-2014.
[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". In Y. Kazakov, D. Lembo and F. Wolter, Eds., 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" Journal of 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://franz.com/agraph/racer/. Last accessed 1-July-2014.
[55] B. Motik, R. Shearer, and I. Horrocks, "Hypertableau reasoning for description logics" Journal of Artificial Intelligence Research, vol. 36, pp. 165–228, 2009.
[56] T. Neumann and G. Weikum, "RDF-3X: a RISC-style engine for RDF" VLDB Endowment, vol.1, no. 1, pp. 647–659, 2008.
[57] T. Neumann and G. Weikum, "Scalable join processing on very large RDF graphs". In U. Çetintemel, S. B. Zdonik, D. Kossmann and N. Tatbul, Eds., Proceedings of the 35th SIGMOD international conference on Management of data - SIGMOD'09, p. 627, 2009, ACM.
[58] T. Neumann and G. Moerkotte, "Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins". In S. Abiteboul, K. Böhm, C. Koch, K. Tan, Eds., Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, pp. 984–994, Apr. 2011.
[59] Y. Ioannidis, "The history of histograms (abridged)", In J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger and A Heuer, Eds., VLDB '03 Proceedings of the 29th international conference on Very large data bases, pp. 19–30, 2003.
[60] W3C, OWL Web Ontology Language Semantics and Abstract Syntax. W3C Recommendation, 2004. [Online]. Available http://www.w3.org/TR/owl-semantics/. Last accessed 1-July-2014.
[61] V. Kolovski, Z. Wu, and G. Eadon, "Optimizing enterprise-scale OWL 2 RL reasoning in a relational database system". In P. F. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Zhang, J. Z. Pan, I. Horrocks and B. Glimm, Eds., 9th International Semantic Web Conference, pp. 436–452, 2010, Springer.
[62] A. Hogan, A. Harth, and A. Polleres, "Scalable authoritative OWL reasoning for the web" International Journal on Semantic Web and Information Systems, vol. 5, no. 2, pp. 49–90, Jan. 2009.
[63] A. Hogan, J. Pan, A. Polleres, and S. Decker, "SAOR: template rule optimisations for distributed reasoning over 1 billion linked data triples". In P. F. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Zhang, J. Z. Pan, I. Horrocks and B. Glimm, Eds., 9th International Semantic Web Conference, 2010, vol. 1380, pp. 337–353.
[64] J. Weaver and J. A. Hendler, "Parallel materialization of the finite RDFS closure for hundreds of millions of triples". In A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta and K. Thirunarayan, Eds., Proceedings of the 8th International Semantic Web Conference, pp. 682–697, 2009, Springer.
[65] J. Urbani, S. Kotoulas, J. Maassen, F. Van Harmelen, and H. Bal, "WebPIE: A Web-scale Parallel Inference Engine using MapReduce" Journal of Web Semantics: Science, Services and Agents on the World Wide Web, vol. 10, pp. 59–75, Jan. 2012.
[66] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems" Journal of Web Semantics, vol. 3, no. 2, pp. 158–182, 2005.
[Food] Food ontology, http://www.w3.org/2001/sw/WebOnt/guide-src/food. Last accessed 1-Jul-2014.
[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. Last accessed 1-Jul-2014.
[Koala] Koala ontology, http://protege.stanford.edu/plugins/owl/owl-library/koala.owl. Last accessed 1-Jul-2014.
[Madcows] Mad cows ontology, http://www.cs.man.ac.uk/~horrocks/OWL/Ontologies/mad_cows.owl. Last accessed 1-Jul-2014.
[Mindswappers] Mindswappers ontology, http://www.mindswap.org/2003/owl/mindswap. Last accessed 1-Jul-2014.
[Minitambis] Mini tambis ontology, http://www.mindswap.org/ontologies/debugging/miniTambis.owl. Last accessed 1-Jul-2014.
[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. Last accessed 1-Jul-2014.
[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. Last accessed 1-Jul-2014.
[Pizza] Pizza ontology, http://www.co-ode.org/ontologies/pizza/pizza_20041007.owl. Last accessed 1-Jul-2014.
[AKT-portal] AKT Portal ontology, http://www.aktors.org/ontology/portal. Last accessed 1-Jul-2014.
[Tambis full] Tambis full ontology, http://www.mindswap.org/ontologies/tambis-full.owl. Last accessed 1-Jul-2014.
[Tap] Tap ontology, http://www.aktors.org/ontology/portal
[Wine] Wine ontology, http://www.w3.org/2001/sw/WebOnt/guide-src/wine. Last accessed 1-Jul-2014.