Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence

# Reasoning and Proofing Services for Semantic Web Agents

**Kalliopi Kravari[1], Konstantinos Papatheodorou[2], Grigoris Antoniou[2] and Nick Bassiliades[1]**

[1]Dept. of Informatics, Aristotle University of Thessaloniki, GR-54124 Thessaloniki, Greece
{kkravari, nbassili}@csd.auth.gr
[2]Institute of Computer Science, FORTH, Greece
Department of Computer Science, University of Crete, Greece
{cpapath, antoniou}@ics.forth.gr

## Abstract

The Semantic Web aims to offer an interoperable environment that will allow users to safely delegate complex actions to intelligent agents. Much work has been done for agents' interoperability; especially in the areas of ontology-based metadata and rule-based reasoning. Nevertheless, the SW proof layer has been neglected so far, although it is vital for agents and humans to understand how a result came about, in order to increase the trust in the interchanged information. This paper focuses on the implementation of third party SW reasoning and proofing services wrapped as agents in a multi-agent framework. This way, agents can exchange and justify their arguments without the need to conform to a common rule paradigm. Via external reasoning and proofing services, the receiving agent can grasp the semantics of the received rule set and check the validity of the inferred results.

## 1 Introduction

The Semantic Web (SW) [Berners-Lee et al., 2001] is a rapidly evolving extension of the current Web, where the semantics of information and services is well-defined, making it possible for people and machines to precisely understand Web content. So far, the fundamental SW technologies (content representation, ontologies) have been established and researchers are currently focusing their efforts on logic and proofs.

Intelligent agents (IAs) are software programs intended to perform tasks more efficiently and with less human intervention. They are considered the most prominent means towards realizing the SW vision [Hendler, 2001]. The gradual integration of multi-agent systems (MAS) with SW technologies will affect the use of the Web in the future; its next generation will consist of groups of intercommunicating agents traversing it and performing complex actions on behalf of users. Thus, IAs are considered to be greatly favored by the interoperability that SW technologies aim to achieve.

IAs will often interact with other agents. However, it is unrealistic to expect that all inter-communicating agents will share a common rule or logic representation formalism; neither can W3C impose specific logic formalisms in a dynamic environment like the Web. Nevertheless, agents should somehow share an understanding of each other's position justification arguments, i.e. logical conclusions based on corresponding rule sets and facts. This heterogeneity in representation and reasoning technologies comprises a critical drawback in agent interoperation.

A solution to this compatibility issue could emerge via equipping each agent with its own inference engine or reasoning mechanism, which would assist in "grasping" other agents' logics. Nevertheless, every rule engine possesses its own formalism and, consequently, agents would require a common interchange language. Since generating a translation schema from one (rule) language into the other (e.g. RIF – Rule Interchange Format) is not always plausible, this approach does not resolve the agent intercommunication issue, but only moves the setback one step further, from argument interchange to rule translation.

An more pragmatic approach was presented in [Kravari et al., 2010a; Kravari et al., 2010b], where reasoning services (called Reasoners) are wrapped in IAs, embedded in a common framework for interoperating SW agents, called EMERALD. This approach allows each agent to effectively exchange its arguments with any other agent, without the need for all involved agents to conform to the same kind of rule paradigm or logic. This way, agents remain lightweight and flexible, while the tasks of inferring know-

ledge from agent rule bases and verifying the results is conveyed to the reasoning services.

Moreover, trust is a vital feature for Semantic Web. If users (humans and agents) are to use and integrate system answers, they must trust them. Thus, systems should be able to explain their actions, sources, and beliefs. Proofing services are extremely important for this purpose as they let users to trust the inference services' results. Traditional trust models (EMERALD supports some of them) are able to guarantee the agents trustworthiness, including the Reasoners' trustworthiness. However, they cannot guarantee the correctness of the inference service itself, meaning that the results exchanged between agents should be explainable to each other. This includes the ability to provide the proof for a certain claim, as a result of an inference procedure, as well as the ability to validate this proof. Therefore, automating proof generation, exchange and validation are important for every inference task in the SW.

As the available inference engines list is constantly expanding, the aim of this paper is to extend EMERALD by adding both new defeasible reasoning and proofing services. In the rest of the paper, we briefly present the EMERALD multi-agent framework, we review some of the reasoners it employs, we describe the new reasoning service for DR-Prolog, and we report on two new services for generating and validating proofs.

## 2 The EMERALD Multi-Agent Framework

EMERALD is a multi-agent knowledge-based framework (Fig. 1), which offers flexibility, reusability and interoperability of behavior between agents, based on Semantic Web and FIPA language standards.

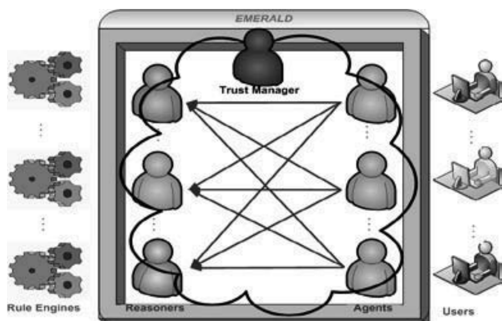



Figure1: *EMERALD Generic Overview.*

In order to model and monitor the parties involved in a transaction, a generic, reusable agent prototype for *knowledge-customizable agents* (*KC-Agents*), consisted of an agent model (*KC Model*), a yellow pages service (*Advanced Yellow Pages Service*) and several external Java methods (*Basic Java Library – BJL*), is deployed (Fig. 2).

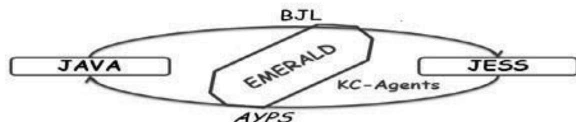



Figure 2: *The KC-Agents Prototype.*

Agents that comply with this prototype are equipped with a Jess rule engine (www.jessrules.com) and a knowledge base (KB) that contains environment knowledge (in the form of facts), behavior patterns and strategies (in the form of Jess production rules). More details can be found in [Kravari et al., 2010a]. The use of the *KC-Agents* prototype offers certain advantages, like interoperability of behavior between agents, as opposed to having behavior hard-wired into the agent's code.

Finally, as agents do not necessarily share a common rule or logic formalism, it is vital for them to find a way to exchange their position arguments seamlessly. Thus, EMERALD proposes the use of *Reasoners* [Kravari et al., 2010b], which are actually agents that offer reasoning services to the rest of the agent community. EMERALD has already been used [Kravari et al., 2011] as a reasoner service provider in SymposiumPlanner, an agent-based conference organization application of the Rule Responder multi-agent platform [Paschke et al., 2007].

## 3 Reasoners

EMERALD's approach for reasoning tasks relies on exchanging the results of the reasoning process of the rule base over the input data. The receiving agent uses an external reasoning service to grasp the semantics of the rulebase, i.e. the set of conclusions of the rule base (Fig. 3). According to the procedure, each Reasoner waits for new requests and when it receives a valid request, it launches the associated reasoning engine and returns the results.

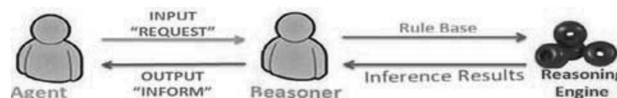



Figure 3: *Input – Output of a Reasoner Agent.*

EMERALD implements Reasoners for two major reasoning formalisms: deductive and defeasible reasoning. *Deductive reasoning* is based on classical logic arguments (modus ponens). *Defeasible reasoning* [Nute, 1987] is a non-monotonic rule-based approach for efficient reasoning with incomplete and inconsistent information. Its main advantages are enhanced representational capabilities and low computational complexity [Maher, 2001].

The two deductive reasoners of EMERALD are the *R-Reasoner* and the *Prova-Reasoner* [Kozlenkov et al., 2006]. *R-Reasoner* is based on *R-DEVICE* [Bassiliades and Vlahavas, 2006], a deductive object-oriented knowledge base system for querying and reasoning about RDF metadata. The system is based on an OO RDF data model, which maps resources to objects and encapsulates properties inside them. R-DEVICE features a RuleML-compatible deductive rule language able to express queries both on the RDF schema and data, including generalized path expressions, stratified negation, aggregate, grouping, and sorting, functions. It uses second-order syntax with variables ranging over class and slot names, which is translated into sets of first-order logic rules using metadata. R-

DEVICE rules define views which are materialized and incrementally maintained, using fixpoint semantics.

The two defeasible reasoners are *DR-Reasoner* and *SPINdle-Reasoner* [Lam and Governatori, 2009]. *DR-Reasoner* is based on the DR-DEVICE system [Bassiliades et al., 2006], which works by accepting as input the address of a defeasible logic rule base, written in an OO RuleML-like syntax. The rule base contains only rules; the facts for the rule program are contained in RDF documents, whose addresses are declared in the rule base. Conclusions are exported as an RDF document. DR-DEVICE is based on the OO RDF model of R-DEVICE, and defeasible rules are implemented through compilation into the generic rule language of R-DEVICE. DR-DEVICE supports multiple rule types of defeasible logic, both classical (strong) negation and negation-as-failure, and conflicting literals, i.e. derived objects that exclude each other.

Reasoners commit to SW and FIPA standards, e.g. RuleML [Boley and Tabet, 2000] for representing and exchanging agent policies and e-contract clauses, since it has become a de facto standard and the RDF model for data representation both for the private data included in agents' internal knowledge and the reasoning results generated during the process. For some of them RuleML support is inherent in the original rule engine (e.g. (D)R-DEVICE), whereas in SPINDLE and Prova, the interface is provided by the wrapper agent.

## 4 DR-Prolog Reasoner

In this work, a new defeasible Reasoner supporting DR-Prolog [Antoniou and Bikakis, 2007] was implemented. DR-Prolog uses rules, facts and ontologies, and supports all major Semantic Web standards, such as RDF/S and RuleML. Moreover, it deals with both monotonic and nonmonotonic rules, open and closed world assumption and reasoning with inconsistencies.

The DR-Prolog Reasoner follows the EMERALD Reasoners' general functionality, waiting for new requests, launching DR-Prolog and returning the results. However, it has to add some new steps in the procedure in order to be able to process the receiving queries and to send back the appropriate answer in RDF format (Fig. 4).

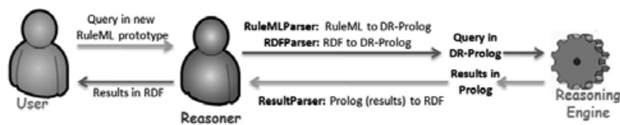
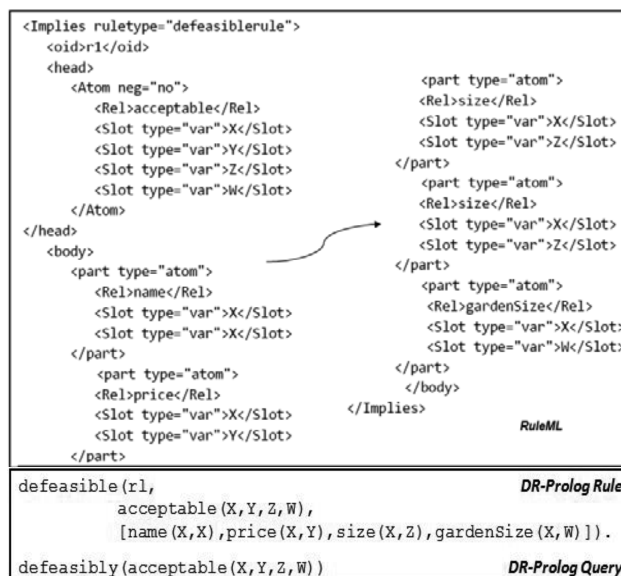



Figure 4: *DR-Prolog Reasoner Functionality.*

The DR-Prolog Reasoner receives a query in an ad-hoc RuleML format. The implemented RuleMLParser receives the RuleML file with the query, extracts the DR-Prolog rules and stores them in native DR-Prolog format. At first the parser processes the rules in the RuleML file, generating the corresponding DR-Prolog rules (Fig. 5). Then, the parser extracts the queries that are included in the RuleML query, indicating whether it is an "answer" or a "proof" query. Fig. 5 shows a query in DR-Prolog format.

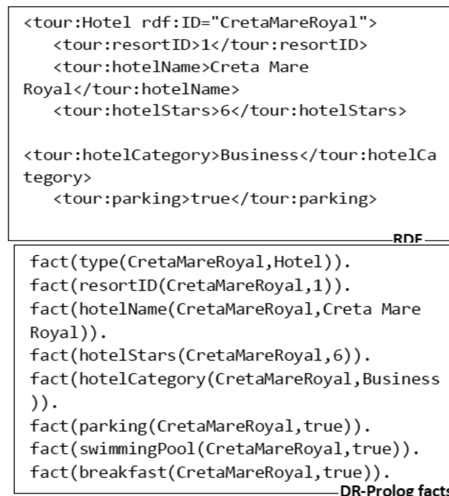


```
<Implies ruletype="defeasiblerule">
  <oid>r1</oid>
  <head>
    <Atom neg="no">
      <Rel>acceptable</Rel>
      <Slot type="var">X</Slot>
      <Slot type="var">Y</Slot>
      <Slot type="var">Z</Slot>
      <Slot type="var">W</Slot>
    </Atom>
</head>
  <body>
    <part type="atom">
      <Rel>name</Rel>
      <Slot type="var">X</Slot>
      <Slot type="var">X</Slot>
    </part>
      <part type="atom">
      <Rel>price</Rel>
      <Slot type="var">X</Slot>
      <Slot type="var">Y</Slot>
    </part>
      <part type="atom">
      <Rel>size</Rel>
      <Slot type="var">X</Slot>
      <Slot type="var">Z</Slot>
    </part>
      <part type="atom">
      <Rel>size</Rel>
      <Slot type="var">X</Slot>
      <Slot type="var">Z</Slot>
    </part>
      <part type="atom">
      <Rel>gardenSize</Rel>
      <Slot type="var">X</Slot>
      <Slot type="var">W</Slot>
    </part>
    </body>
  </Implies>
RuleML

defeasible(r1,                                                DR-Prolog Rule
        acceptable(X,Y,Z,W),
        [name(X,X),price(X,Y),size(X,Z),gardenSize(X,W)]).
defeasibly(acceptable(X,Y,Z,W))                               DR-Prolog Query
```

Figure 5: *RuleMLParser: RuleML to DR-Prolog.*

However, turning the initial RuleML query and rulebase into DR-Prolog is not enough. The fact base has to be translated, too. Typically, the fact base is in RDF format, which must be transformed into Prolog facts. For this purpose, another parser was implemented, called RDFParser. This parser uses the SW Knowledge Middleware, a set of tools for the parsing, storage, manipulation and querying of Semantic Web (RDF) Knowledge bases, to extract the RDF triples and turn them to Prolog facts (Fig. 6).



```
<tour:Hotel rdf:ID="CretaMareRoyal">
   <tour:resortID>1</tour:resortID>
   <tour:hotelName>Creta Mare
Royal</tour:hotelName>
   <tour:hotelStars>6</tour:hotelStars>

<tour:hotelCategory>Business</tour:hotelCa
tegory>
   <tour:parking>true</tour:parking>
RDF

fact(type(CretaMareRoyal,Hotel)).
fact(resortID(CretaMareRoyal,1)).
fact(hotelName(CretaMareRoyal,Creta Mare
Royal)).
fact(hotelStars(CretaMareRoyal,6)).
fact(hotelCategory(CretaMareRoyal,Business
)).
fact(parking(CretaMareRoyal,true)).
fact(swimmingPool(CretaMareRoyal,true)).
fact(breakfast(CretaMareRoyal,true)).
DR-Prolog facts
```

Figure 6: *RDFParser: RDF rule base to DR-Prolog facts.*

Thus, a new query in DR-Prolog with the associated rule base in Prolog facts is available. The DR-Prolog Reasoner invokes the inference engine and when the inference results (in Prolog) are available the Reasoner transforms them into RDF format and forwards them back to the requesting agent. For this purpose, a third parser called ResultParser was implemented. This parser receives the initial query (in DR-Prolog) and the results (a prolog list) and

returns the query results in RDF (Fig. 7). It is important to mention that the returned RDF results contain only the results that are required by the initial query and not the complete information that is available at the results' base.

```
<dr-device:acceptable rdf:about="#acceptable0">
      <dr-device:X>a3</dr-device:X>
      <dr-device:Y>350</dr-device:Y>
      <dr-device:Z>65</dr-device:Z>
      <dr-device:W>0</dr-device:W>
</dr-device:acceptable>
```

Figure 7: *ResultParser: Prolog to RDF.*

# 5 Defeasible Proofing Services

The Proof layer of the SW is assumed to answer agents about the question of why they should believe the results. At present, there is no technology recommended by W3C to this layer. However, it is a vital issue and thus researchers are now focusing their attention on this direction.

The DR-Prolog Reasoner that was presented in the previous section was updated with two new services: a) a proofing service that provides proof explanations (for the inference results), increasing the trust of the users, and b) a proof validation service, that validates proofs provided by other agents.

However, both services are available only on demand; meaning that the Reasoner can also be used as usual sending back just the results; but whenever it is requested to provide explanations (proofs) or check provided proofs, the service uses its corresponding proofing service and sends back both the results and their explanation, along with their validity reassurance. More about the project teams that worked in both DR-Prolog Reasoner and Proof Validator, can be found at [CS-566 Project, 2010].

## 5.1 Defeasible Proof Generator Service

A defeasible proof explanation functionality was added to the DR-Prolog Reasoner. At first, the Reasoner was equipped with a proof generator [Antoniou et al., 2008], which produces automatically proof explanations using the XSB logic programming system, by interpreting the output from the proof's trace and converting it into a meaningful representation. This proof generator supports explanations in defeasible logic for both positive and negative answers in user queries. Additionally, it uses a pruning algorithm that reads the XSB trace and removes the redundant information in order to formulate a sensible proof.

## 5.2 Defeasible Proof Validator Service

The DR-Prolog Reasoner was also equipped with the functionality of a defeasible proof validator, which gives the ability to check proofs provided by other agents. The provided proof explanations are fed into the proof validator that verifies the validity of the conclusions.

The Proof Validator follows the Reasoners' functionality, i.e. it receives requests and returns back a reply; however, it can also be embedded in any other system. Its competence is to decide whether the provided proof, given a theory (in XML), is valid or not. In the case the proof is not valid, an appropriate error message is returned, depending on the nature of the problem.

At first, the proof validator receives the validation request, then parses the received proof and constructs a prolog query based on the claimed proof. The received theory is also constructed and loaded on the prolog engine in support to the proof validator. The XML-formatted proof, contains a lot of redundant or unnecessary information that is omitted from the validation query. Elements declaring provability of a predicate along with the predicate and the rule name are more important, while elements describing the body of a rule or superiority claims are ignored. The validator also uses DR-Prolog.

Four assumptions were made during the implementation:

1. The theory is the one given to the proof generator. Any given theory is accepted as valid without any checks.
2. No checks are performed recursively. Thus, any information is required in depth more than one a priori.
3. Any knowledge (facts) given in the theory is considered to be definitely provable, not taking into account statements present in the proof supporting it.
4. The minimal information that will contribute to the proof checking process is required.

**Data Structures**

The proof validator features two groups of collections, theory structures and proof deduction structures. The theory structures are four knowledge bases holding the strict rules, defeasible rules, facts and rules hierarchy.

The proof deduction structures are knowledge bases holding any deduced information already stated by the proof and confirmed by the validator. For example, a rule that adds knowledge to the definite KB is the following:

```
definitelyCheck(X, printOn) :-
    factkb(F),memberchk(X,F),addDefinitely(X).
```

**Facts**

All facts given in the theory are also added in the definite knowledge base. This means that any stated fact is by default considered by the proof validator as definitely proved.

**Rules**

Generally the rules are not stated explicitly in the contents of the proof validation result, since they are a priori accepted and considered valid. More specifically in the proof, the rules are not stated explicitly, a mere reference of the rule name is used: for example defeasibly(e,r2) in the proof means that e is derived using rule r2 which can be found in the rule base of the original theory.

**Deductions**

A conclusion in D (defeasible) is a tagged literal and may have one of the following forms [Antoniou et al., 2001]:

- $+\Delta q/+\partial q$: q is definitely/defeasibly provable in D.
- $-\Delta q/-\partial q$: q has proved to be not definitely/defeasibly provable in D.

In order to prove $+\Delta q$, a proof for q consisting of facts and strict rules needs to be established. Whenever a literal

is definitely provable, it is also defeasibly provable. In that case, the defeasible proof coincides with the definite proof for q. Otherwise, in order to prove $+\partial q$ in D, an applicable strict or defeasible rule supporting q must exist. In addition, it should also be ensured that the specified proof is not overridden by contradicting evidence. Therefore, it has to be guaranteed that the negation of q is not definitely provable in D. Successively, every rule that is not known to be inapplicable and has head ~q has to be considered. For each such rule s, it is required that there is a counterattacking rule t with head q that is applicable at this point and s is inferior to t.

In order to prove $-\Delta q$ in D, q must not be a fact and every strict rule supporting q must be known to be inapplicable. If it is proved that $-\Delta q$, then it is also proved that $-\partial q$. Otherwise, in order to prove that $-\partial q$, it must firstly be ensured that $-\Delta q$. Add itionally, one of the following conditions must hold: (i) None of the rules with head q can be applied, (ii) It is proved that $-\Delta$~q, and (iii) There is an applicable rule r with head ~q, such that no possibly applicable rule s with head q is superior to r.

The proof validator, presented here, uses two separate predicates for strict deductions, namely $+\Delta$ (definitely) and $-\Delta$ (not definitely) and two for defeasible deductions; predicates defeasibly for $+\partial$ and not defeasibly for $-\partial$. In general, for positive deductions ($+\Delta$ and $+\partial$), when the conclusion is proven by a rule, the name of the rule is required by the proof validator. Otherwise, i.e. when the conclusion is a fact, or when it is already given or deducted at a previous step, it is not required. This approach is followed for the sake of efficiency. On the other hand, for negative deductions, giving the name of the rule would be redundant, because even if a negative result is concluded by one rule, the validator still has to retrieve all existing relevant rules regardless of whether the proof states them or not. For example, the rules that checks if a stated literal, claimed to be either a fact or an already proven literal, is definitely provable or not, are the following:

```
definitelyCheck(X,printOn) :-
   factkb(F),memberchk(X,F),addDefinitely(X).
definitelyCheck(X,printOff) :-
   factkb(F), memberchk(X,F).
definitelyCheck(X,_) :-
   definitelykb(K), memberchk(X,K).
definitelyCheck(X,Print) :-
   logError(Print,[X,' ……..']).
```

The second argument is used to state if errors are to be printed or not. The first two rules check if X is a fact. The third rule checks if X is already a member of the definite knowledge base. Should these three rules fail, it means that X is neither a fact nor a literal that has been proven definitely, so the fourth rule prints an error message.

**Examples**

Below the evaluation steps for a complex team defeat example are explained.

```
r1: a => e.        r2: b => e.
r3: c => ~e.       r4: d => ~e.
r1 > r3.     r2 > r4.     a. b. c. d.
```

A valid and correct proof is the following:
defeasibly(a), defeasibly(b), defeasibly(c), defeasibly(d), defeasibly(e, r2).

The first four statements are deduced in an obvious way, since a, b, c, d are facts. The last, i.e. defeasibly(e,r2), has to make some proof checks such as:

1. Is there any rule "r2" in the theory, with head equal to e? → Yes.
2. Is there any attacking rule? → Yes, r3, r4
   2.1 Is r2 of higher priority than r3? → No.
      2.1.1 Are the conditions of r3 (i.e. c) defeasibly provable? → Yes.
      2.1.2 Is there any attacking rule of r3 (different from r2), which defeats r3? → Yes, r1, because its conditions are met and r1 > r3.
   2.2 Is r2 of higher priority than r4? → Yes.

# 6 Conclusions and Future Work

The paper argued that agents will play a vital role in the realization of the SW vision and presented a variety of reasoning services called Reasoners, wrapped in an agent interface, embedded in a common framework for interoperating SW IAs, called *EMERALD*, a JADE multi-agent framework designed specifically for the Semantic Web. This methodology allows each agent to effectively exchange its argument base with any other agent, without the need for all involved agents to conform to the same kind of rule paradigm or logic. Instead, via EMERALD, IAs can utilize third-party reasoning services, that will infer knowledge from agent rule bases and verify the results.

The framework offers a variety of popular inference services that conform to various types of logics. The paper presents how new types of logic were embedded in new Reasoners, as well as it argues about the importance of the SW proof layer and presents how proofing services were designed and embedded also in the system.

A similar architecture for IAs is presented in [Wang et al., 2005], where various reasoning engines are employed as plug-in components, on top of the OPAL agent platform [Purvis et al., 2002]. The primary difference with EMERALD lies in the tight coupling of the rule engines with the agents of the platform, through bindings in the host programming language, whereas EMERALD offers a completely open and loosely-coupled approach, based on first-class agents as reasoning services and on SW standards, for rule and data interchange.

As for future directions, it would be interesting to integrate a broader variety of reasoning and proof validation engines and to develop methodologies to integrate the generated proofs with the trust mechanism of EMERALD, in order to interconnect the Proof and Trust layers of the SW.

# References


[Hendler, 2001] J. Hendler. Agents and the Semantic Web. *IEEE Intelligent Systems*, 16(2):30-37, 2001.

[Kravari et al., 2010a] K. Kravari, E. Kontopoulos, and N. Bassiliades. EMERALD: A Multi-Agent System for

Knowledge-based Reasoning Interoperability in the Semantic Web. *In Proceedings of the 6th Hellenic Conference on Artificial Intelligence (SETN 2010)*, LNCS, Vol. 6040, pp. 173-182, Springer, 2010a.

[Kravari et al., 2010b] K. Kravari, E. Kontopoulos, and N. Bassiliades. Trusted Reasoning Services for Semantic Web Agents. *Informatica, International Journal of Computing and Informatics*, 34(4):429-440, 2010b.

[Bassiliades et al., 2006] N. Bassiliades, G. Antoniou, and I. Vlahavas. A Defeasible Logic Reasoner for the Semantic Web. *IJSWIS* 2(1):1-41, 2006.

[Boley and Tabet, 2000] H. Boley, and S. Tabet. RuleML: The RuleML Standardization Initiative, http://www.ruleml.org/, 2000.

[Berners-Lee et al., 2001] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American Magazine*, 284(5):34-43, 2001. (Revised 2008).

[Wang et al., 2005] M. Wang, M. Purvis, and M. Nowostawski. An Internal Agent Architecture Incorporating Standard Reasoning Components and Standards-based Agent Communication. *In Proceedings of IAT'05*, 58-64, Washington, DC, IEEE Computer Society, 2005.

[Purvis et al., 2002] M. Purvis, S. Cranefield, M. Nowostawski, and D. Carter. Opal: A Multi-Level Infrastructure for Agent-Oriented Software Development. Information Science Discussion Paper Series, number 2002/01, ISSN 1172-602, University of Otago, Dunedin, New Zealand, 2002.

[Nute, 1987] D. Nute. Defeasible Reasoning. *In Proceedingsof the 20th International Conference on Systems Science*, 470-477. IEEE, 1987.

[Maher, 2001] M.J. Maher. Propositional defeasible logic has linear complexity. *Theory and Practice of Logic Programming*, 1(6):691–711, 2001.

[Bassiliades and Vlahavas, 2006] N. Bassiliades, and I. Vlahavas. R-DEVICE: An Object-Oriented Knowledge Base System for RDF Metadata. *International Journal on Semantic Web and Information Systems*, 2(2):24-90, 2006.

[Antoniou and Bikakis, 2007] G. Antoniou, and A. Bikakis. DR-Prolog: A System for Defeasible Reasoning with Rules and Ontologies on the SW. *IEEE Transactions on Knowledge and Data Engineering*, 19,2, 2007.

[Antoniou et al, 2008] G. Antoniou, A. Bikakis, N. Dimaresis, and G. Governatori. Proof Explanation for a Nonmonotonic Semantic Web Rules Language. *Data and Knowledge Engineering*, 64(3)662-687, 2008.

[CS-566 Project, 2010] CS-566 Project, http://www.csd.uoc.gr/~hy566/ project2010.html, 2010.

[Antoniou et al, 2001] G. Antoniou, D. Billington, G. Governatori, and M.J. Maher. Representation results for defeasible logic. *ACM Trans. Comput. Logic*, 2(2):255–287, 2001.

[Lam and Governatori, 2009] H. Lam, and G. Governatori. The Making of SPINdle. *International Symposium on Rule Interchange and Applications (RuleML-2009)*, pp. 315-322, Springer, 2009.

[Kozlenkov et al., 2006] A. Kozlenkov, R. Penaloza, V. Nigam, L. Royer, G. Dawelbait, and M. Schroeder. Prova: Rule-based Java Scripting for Distributed Web Applications: A Case Study in Bioinformatics. Reactivity on the Web Workshop, Munich, 2006.

[Kravari et al., 2011] K. Kravari, T. Osmun, H. Boley, and N. Bassiliades. Cross-Community Interoperation Between the EMERALD and Rule Responder Multi-Agent Systems. *In Proceedings of the 5th International Symposium on Rules: Research Based and Industry Focused (RuleML-2011) co-located with IJCAI-11*, Barcelona, Spain, 19-21 July 2011.

[Paschke et al., 2007] A. Paschke, H. Boley, A. Kozlenkov, and B. Craig. Rule responder: RuleML-based Agents for Distributed Collaboration on the Pragmatic Web. *In Proceedings of the 2nd International Conference on Pragmatic Web*, 17-28, vol. 280. ACM, 2007.