# Trusted Reasoning Services for Semantic Web Agents

Kalliopi Kravari, Efstratios Kontopoulos and Nick Bassiliades
Dept. of Informatics, Aristotle University of Thessaloniki, Greece, GR-54124
E-mail: {kkravari, skontopo, nbassili}@csd.auth.gr

**Keywords:** semantic web, intelligent agents, multi-agent system, reasoning

**Received:** January 16, 2010

*The Semantic Web aims at enriching information with well-defined semantics, making it possible both for people and machines to understand Web content. Intelligent agents are the most prominent approach towards realizing this vision. Nevertheless, agents do not necessarily share a common rule or logic formalism, neither would it be realistic to attempt imposing specific logic formalisms in a rapidly changing world like the Web. Thus, based on the plethora of proposals and standards for logic- and rule-based reasoning for the Semantic Web, a key factor for the success of Semantic Web agents lies in the interoperability of reasoning tasks. This paper reports on the implementation of trusted, third party reasoning services wrapped as agents in a multi-agent system framework. This way, agents can exchange their arguments, without the need to conform to a common rule or logic paradigm – via an external reasoning service, the receiving agent can grasp the semantics of the received rule set. Finally, a use case scenario is presented that illustrates the viability of the proposed approach.*

*Povzetek: Semantični spletni agenti potrebujejo oceno zaupanja storitev za kvalitetno delovanje.*

## 1 Introduction

The *Semantic Web* (*SW*) is a rapidly evolving extension of the World Wide Web that derives from Sir Tim Berners-Lee's vision of a universal medium for data, information and knowledge exchange [1]. The SW aims at augmenting Web content with well-defined semantics (i.e. meaning), making it possible both for people and machines to comprehend the available information and better satisfy their requests. 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* (*IA*s – software programs extended to perform tasks more efficiently and with less human intervention) are considered the most prominent means towards realizing the SW vision [2]. The gradual integration of *multi-agent systems* (*MAS*) with SW technologies will affect the use of the Web in the imminent future; its next generation will consist of groups of intercommunicating agents traversing it and performing complex actions on behalf of their users.

IAs, on the other hand, are considered to be greatly favored by the interoperability that SW technologies aim to achieve. Thus, IAs will often interact with other agents, belonging to service providers, e-shops, Web enterprises or even other users. However, it is unrealistic to expect that all intercommunicating agents will share a common rule or logic representation formalism; neither can W3C impose specific logic formalisms in a drastically dynamic environment like the Web. In order for agent interactions to be meaningful, 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* [3]) 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/transformation.

An alternative, more pragmatic, approach is presented in this work, where reasoning services are wrapped in IAs. Although we have embedded these reasoners in a common framework for interoperating SW agents, called *EMERALD*[1], they can be added in any other multi-agent system. The motivation behind this approach is to avoid the drawbacks outlined above and propose utilizing third-party reasoning services, instead, that allow 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 knowledge from agent rule bases and verifying the results is conveyed to the reasoning services.

Flexibility is a key aim for our research, thus a variety of popular inference services that conform to various

[1] *http://lpis.csd.auth.gr/systems/emerald/emerald.html*

types of logics is offered and the list is constantly expanding. Furthermore, the notion of trust is vital, since agents need a mechanism for establishing trust towards the reasoning services, so that they can trust the generated inference results. Towards this direction, reputation mechanisms (centralized and decentralized) were proposed and integrated in the EMERALD framework.

The rest of the paper is structured as follows: Section 2 presents a brief overview of the framework, followed by a more thorough description of the reasoning services, in Section 3. Section 4 features the implemented trust mechanisms, while Section 5 reports on a brokering use case scenario that illustrates the use of the reasoning services and the reputation methodology. Finally, the paper is concluded with an outline of related work paradigms, as well as the final remarks and directions for future improvements.

## 2 Framework overview

The EMERALD framework is built on-top of JADE[2] and, as mentioned in the introduction, it involves trusted, third-party reasoning services, deployed as agents that infer knowledge from an agent's rule base and verify the results. The rest of the agents can communicate with these services via ACL message exchange.

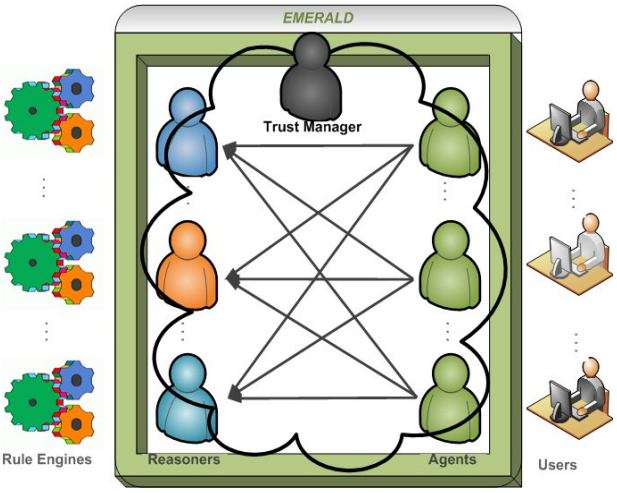


Figure 1: Generic Overview.

Figure 1 illustrates a generic overview of the framework: each human user controls a single all-around agent; agents can intercommunicate, but do not have to "grasp" each other's logic. This is why third-party, reasoning services are deployed. In our approach, reasoning services are "wrapped" by an agent interface, called the Reasoner (presented later), allowing other agents to contact them via *ACL* (*Agent Communication Language*) messages.

The element of trust is also vital, since an agent needs to trust the inference results returned from a Reasoner and is established via centralized and decentralized reputation mechanisms integrated in EMERALD. Figure 1 displays the aspect of the former (centralized) mechanism, where a specialized "*Trust Manager*" agent keeps the reputation scores for the reasoning services given from the rest of the IAs.

Overall, the goal is to apply as many standards as possible, in order to encourage the application and development of the framework. Towards this affair, a number of popular rule engines that comply with various types of (monotonic and non-monotonic) logics are featured in EMERALD (see section 3). Additionally, *RDF*/*S* (*Resource Description Framework/Schema*) and *OWL* (*Web Ontology Language*) serve as language formalisms, using in practice the Semantic Web as infrastructure for the framework.

## 3 Reasoning services

EMERALD currently implements a number of Reasoner agents that offer reasoning services in two main formalisms: deductive and defeasible reasoning.
Table **1** displays the main features of the reasoning engines described in the following sections.

Table 1: Reasoning engine features.

| | **Type of logic** | **Implementation** |
|---|---|---|
| **R-DEVICE** | deductive | RDF/CLIPS/RuleML |
| **Prova** | deductive | Prolog/Java |
| **DR-DEVICE** | defeasible | RDF/CLIPS/RuleML |
| **SPINdle** | defeasible | XML/Java |
| | **Order of Logic** | **Reasoning** |
| **R-DEVICE** | 2nd order | fwd chaining |
| **Prova** | 1st order | bwd chaining |
| **DR-DEVICE** | 2nd order | fwd chaining |
| **SPINdle** | 1st order | fwd chaining |

*Deductive reasoning* is based on classical logic arguments, where conclusions are proved to be valid, when the premises of the argument (i.e. rule conditions) are true. *Defeasible reasoning* [4], on the other hand, constitutes a non-monotonic rule-based approach for efficient reasoning with incomplete and inconsistent information. When compared to more mainstream non-monotonic reasoning approaches, the main advantages of defeasible reasoning are enhanced representational capabilities and low computational complexity [5]. The following subsection gives a brief insight into the fundamental elements of defeasible logics.

### 3.1 Defeasible logics

A *defeasible theory D* (i.e. a knowledge base or a program in defeasible logic) consists of three basic ingredients: a set of facts ($F$), a set of rules ($R$) and a superiority relationship ($>$). Therefore, $D$ can be represented by the triple ($F$, $R$, $>$).

In defeasible logic, there are three distinct types of rules: strict rules, defeasible rules and defeaters. *Strict rules* are denoted by $A \rightarrow p$ and are interpreted in the typical sense: whenever the premises are indisputable, so

[2] *JADE (Java Agent Development Environment): http://jade.tilab.com/*

is the conclusion. An example of a strict rule is: "*Apartments are houses*", which, written formally, would become: `r1: apartment(X) → house(X).`

*Defeasible rules* are rules that can be defeated by contrary evidence and are denoted by $A \Rightarrow p$. An example of such a rule is "*Any apartment is considered to be acceptable*", which becomes: `r2: apartment(X)` $\Rightarrow$ `acceptable(X).`

*Defeaters*, denoted by $A \in p$, are rules that do not actively support conclusions, but can only prevent some of them. In other words, they are used to defeat some defeasible rules by producing evidence to the contrary. An example of a defeater is: `r3: pets(X), gardenSize(X,Y), Y>0` $\in$ `acceptable(X)`, which reads as: "*If pets are allowed in the apartment, but the apartment has a garden, then it might be acceptable*". This defeater can defeat, for example, rule `r4: pets(X)` $\Rightarrow$ $\neg$`acceptable(X).`

Finally, the *superiority relationship* among the rule set $R$ is an acyclic relation > on $R$. For example, given the defeasible rules `r2` and `r4`, no conclusive decision can be made about whether the apartment is acceptable or not, because rules `r2` and `r4` contradict each other. But if a superiority relation > with `r4` > `r2` is introduced, then `r4` overrides `r2` and we can indeed conclude that the apartment is considered unacceptable. In this case rule `r4` is called *superior* to `r2` and `r2` *inferior* to `r4`.

Another important element of defeasible reasoning is the notion of *conflicting literals*. In applications, literals are often considered to be conflicting and at most one of a certain set should be derived. An example of such an application is price negotiation, where an offer should be made by the potential buyer. The offer can be determined by several rules, whose conditions may or may not be mutually exclusive. All rules have `offer(X)` in their head, since an offer is usually a positive literal. However, only one offer should be made. Therefore, only one of the rules should prevail, based on superiority relations among them. In this case, the conflict set is:

$$C(offer(x,y)) = \{\neg offer(x,y)\} \cup \{offer(x,z) \mid z \neq y\}$$

For example, the following two rules make an offer for a given apartment, based on the buyer's requirements. However, the second one is more specific and its conclusion overrides the conclusion of the first one.

```
r5: size(X,Y),Y≥45,garden(X,Z)
    ⇒ offer(X,250+2Z+5(Y-45))
r6: size(X,Y),Y≥45,garden(X,Z),central(X)
    ⇒ offer(X,300+2Z+5(Y-45))
r6 > r5
```

## 3.2 Deductive reasoners

EMERALD currently deploys two deductive reasoners, based on the logic programming paradigm: *R-Reasoner* and *Prova-Reasoner*, which deploy the R-DEVICE and Prova rule engines, respectively.

### 3.2.1 R-DEVICE

*R-DEVICE* [6] is 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 is different from the established triple-based model, in the sense that resources are mapped to objects and properties are encapsulated inside resource objects, as traditional OO attributes. More specifically, R-DEVICE transforms RDF triples into CLIPS (COOL) objects and uses a deductive rule language for querying and reasoning about them, in a forward-chaining Datalog fashion. This transformation leads to fewer joins required for accessing the properties of a single resource, subsequently resulting in better inference/querying performance.

Furthermore, R-DEVICE features a deductive rule language (in OPS5/CLIPS-like format or in a RuleML-like syntax) for reasoning on top of RDF metadata. The language supports a second-order syntax, which is efficiently translated into sets of first-order logic rules using metadata, where variables can range over classes and properties, so that reasoning over the RDF schema can be performed. A sample rule in the CLIPS-like syntax is displayed below:

```
(deductiverule test-rule
  ?x <- (website (dc:title ?t) (dc:creator
  "John Smith"))
  =>
  (result (smith-creations ?t))
)
```

Rule `test-rule` above seeks for the titles of websites (class website) created by "`John Smith`". Note that namespaces, like DC, can also be used.

The semantics of the rule language of R-DEVICE are similar to Datalog [7] with a semi-naive evaluation proof procedure and an OO syntax in the spirit of F-Logic [8]. The proof procedure of R-DEVICE dictates that when the condition of the rule is satisfied, then the conclusion is derived and the corresponding object is materialized (asserted) in the knowledge base. R-DEVICE supports non-monotonic conclusions. So, when the condition of a rule is falsified (after being satisfied), then concluded object is retrieved (retracted). R-DEVICE also supports negation-as-failure.

### 3.2.2 Prova

*Prova* [9] is a rule engine for rule-based Java scripting, integrating Java with derivation rules (for reasoning over ontologies) and reaction rules (for specifying reactive behaviors of distributed agents). Prova supports rule interchange and rule-based decision logic, distributed inference services and combines ontologies and inference with dynamic object-oriented programming.

As a declarative language with derivation rules, Prova features a Prolog syntax that allows calls to Java methods, thus, merging a strong Java code base with Prolog features, such as backtracking. For example, the following Prova code fragment features a rule, whose body consists of a number of Java method calls:

```
hello(Name):-
       S = java.lang.String("Hello "),
       S.append(Name),
       java.lang.System.out.println(S).
```

On the other hand, Prova reaction rules are applied in specifying agent behavior, leaving more critical operations (e.g. agent messaging etc.) to the language's Java-based extensions. In this affair, various communication frameworks can be deployed, like JADE, JMS[3] or even Java events generated by Swing (G.U.I.) components. Reaction rules in Prova have a blocking `rcvMsg` predicate in the head and fire upon receipt of a corresponding event. The `rcvMsg` predicate has the following syntax: `rcvMsg(Protocol, To, Performative, [Predicate|Args] | Context)`. The following code fragment shows a simplified reaction rule for the FIPA *queryref* performative:

```
rcvMsg(Protocol,From,queryref,[Pred|Args]|
Context):-
   derive([Pred|Args]),
   sendMsg(Protocol,From,reply,[Pred|Args]
   |Context).
rcvMsg(Protocol,From,queryref,[Pred|Args],
Protocol):-
   sendMsg(Protocol,From,end_of_transmissi
   on,[Pred|Args]|Context).
```

The `sendMsg` predicate is embedded into the body of derivations or reaction rules and fails only if the parameters are incorrect or if the message could not be sent due to various other reasons, like network connection problems. Both code fragments presented above were adopted from [9].

Prova is derived from *Mandarax* [10], an older Java-based inference engine, and extends it by providing a proper language syntax, native syntax integration with Java, agent messaging and reaction rules.

## 3.3 Defeasible reasoners

Furthermore, EMERALD also supports two defeasible reasoners: *DR-Reasoner* and *SPINdle-Reasoner*, which deploy DR-DEVICE and SPINdle, respectively.

### 3.3.1 DR-DEVICE

*DR-DEVICE* [11] is a defeasible logic reasoner, based on R-DEVICE presented above. DR-DEVICE is capable of reasoning about RDF metadata over multiple Web sources using defeasible logic rules. More specifically, the system accepts as input the address of a defeasible logic rule base. 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. After the inference, conclusions are exported as an RDF document. Furthermore, DR-DEVICE supports all defeasible logic features, like rule types, rule superiorities etc., applies two types of negation (strong, negation-as-failure) and conflicting (mutually exclusive) literals.

Similarly to R-DEVICE, rules can be expressed either in a native CLIPS-like language, or in a (further) extension of the OORuleML syntax, called *DR-RuleML*, that enhances the rule language with defeasible logic elements. For instance, rule $r_2$ from section 3.1 can be represented in the CLIPS-like syntax as:

```
(defeasiblerule r2
  (apartment (name ?X))
  =>
  (acceptable (name ?X)))
```

For completeness, we also include the representation of rule $r_4$ from section 3.1 in the CLIPS-based syntax, in order to demonstrate rule superiority and negation:

```
(defeasiblerule r4
  (declare (superior r2))
  (apartment (name ?X) (pets "no"))
  =>
  (not (acceptable (name ?X))))
```

The reasoner agent supporting DR-DEVICE is *DR-Reasoner* [12].

### 3.3.2 SPINdle

*SPINdle* [13] is an open-source, Java-based defeasible logic reasoner that supports reasoning on both standard and modal defeasible logic. It accepts defeasible logic theories, represented via a text-based pre-defined syntax or via a custom XML vocabulary, processes them and exports the results via XML. More specifically, SPINdle supports all the defeasible logic features (facts, strict rules, defeasible rules, defeaters and superiority relationships), modal defeasible logics [14] with modal operator conversions, negation and conflicting (mutually exclusive) literals.

A sample theory that follows the pre-defined syntax of SPINdle is displayed below (adopted from the SPINdle website[4]):

```
>> sh          #Nanook is a Siberian husky.
R1: sh -> d    #Huskies are dogs.
R2: sh => -b   #Huskies usually do not bark.
R3: d => b     #Dogs usually bark.
R2 > R3        #R2 is more specific than R3.
#Defeasibly, Nanook should not bark.
#That is, +d -b
```

Additionally, as a standalone system, SPINdle also features a visual theory editor for editing standard (i.e. non-modal) defeasible logic theories.

## 3.4 Reasoner functionality

The reasoning services, as already mentioned, are wrapped by an agent interface, the *Reasoner*, allowing other IAs to contact them via ACL messages. The Reasoner can launch an associated reasoning engine, in order to perform inference and provide results. In essence, the Reasoner is a service and not an autonomous agent; the agent interface is provided in order to integrate Reasoner

[3] *JMS (Java Message Service): http://java.sun.com/products/jms/*

[4] *http://spin.nicta.org.au/spindleOnline/index.html*

agents into EMERALD or even any other multi-agent system.

The procedure is straightforward (Figure 2): each Reasoner constantly stands by for new requests (ACL messages with a "*REQUEST*" communication act). As soon as it gets a valid request, it launches the associated reasoning engine that processes the input data (i.e. rule base) and returns the results. Finally, the Reasoner returns the above result through an "*INFORM*" ACL message.

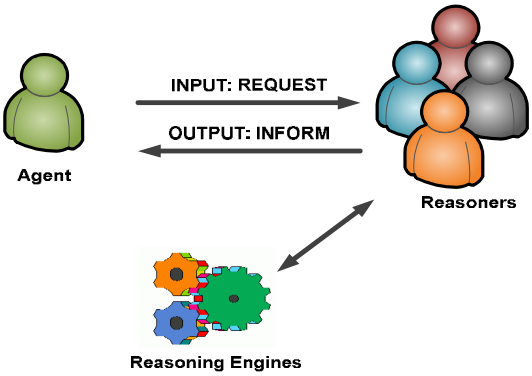


Figure 2: Reasoners' functionality.

A sample ACL message, based on Fipa2000[5] description, in the CLIPS-like syntax is displayed below:

```
(ACLMessage
    (communicative-act REQUEST)
    (sender AgentA@xx:1099/JADE)
    (receiver xx-Reasoner@xx:1099/JADE)
    ....
    (protocol protocolA)
    (language "English")
    (content C:\\rulebase.ruleml)
)
```

where AgentA sends to a Reasoner (`xx-Reasoner`) a RuleML file path (`C:\\rulebase.ruleml`).

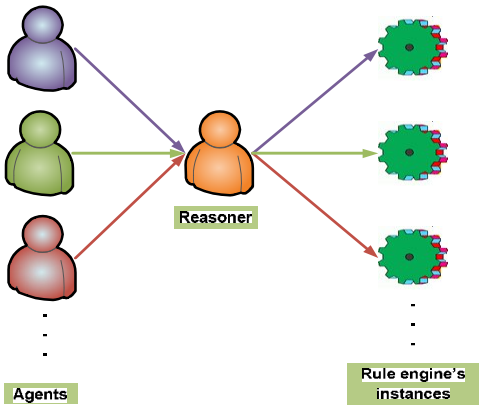


Figure 3: Serving multiple requests.

An important feature of the procedure is that whenever a Reasoner receives a new valid request, it launches a new instance of the associated reasoning engine. Therefore, multiple requests are served concurrently and independently (see Fig. 3). As a result, new requests are served almost immediately, avoiding burdening the framework's performance, because the only sequential operation of the reasoner is the transfer of requests and results between reasoning engines and the requesting agents, which are very low demanding in time.

Finally, note that Reasoners do not use a particular rule language. They simply transfer file paths (in the form of Java Strings) via ACL messages either from a requesting agent to a rule engine or from the rule engine to the requesting agent. Obviously, the content of these files has to be written in the appropriate rule language. For instance an agent who wants to use either the DR-DEVICE or the R-DEVICE rule engine has to provide valid RuleML files. Similarly, valid Prova or XML files are required by the Prova and SPINdle rule engine, respectively. Hence, it is up to the requesting agent's user to provide the appropriate files, by taking each time into consideration the rule engines' specifications.

Thus, new reasoners can be easily created and added to the platform by building a new agent that manages messages between the requesting agent and the rule engine. Furthermore, it has to launch instances of the rule engine according to the specific requirements of the engine.

## 4 Trust mechanisms

Tim Berners-Lee described trust as a fundamental component of his vision for the Semantic Web [1], [15], [16]. Thus, it is not surprising that trust is considered critical for effective interactions among agents in the Semantic Web, where agents have to interact under uncertain and risky situations. However, there is still no single, accepted definition of trust within the research community, although it is generally defined as the expectation of competence and willingness to perform a given task. Broadly speaking, trust has been defined in various ways in literature, depending on the domain of use. Among these definitions, there is one that can be used as a reference point for understanding trust, provided by Dasgupta [17]: "*Trust is a belief an agent has that the other party will do what it says it will (being honest and reliable) or reciprocate (being reciprocative for the common good of both), given an opportunity to defect to get higher payoffs.*"

There are various trust metrics, some involving past experience, some giving relevance to opinions held by an agent's neighbours and others using only a single agent's own previous experience. During the past decade, many different metrics have been proposed, but most have not been widely implemented. Five such metrics are described in [18], among them *Sporas* [19] seems to be the most used metric, although *CR* (*Certified Reputation*) [20] is one of the most recently proposed methodologies.

Our approach adopts two reputation mechanisms, a decentralized and a centralized one. Notice that in both approaches newcomers start with a neutral value. Otherwise, if their initial reputation is set too low, it may be rather difficult to prove trustworthiness through one's

[5] *Fipa2000 description for the ACL Message parameters: www.fipa.org*

actions. If, on the other hand, the reputation is set too high, there may be a need to limit the possibility for users to "start over" after misbehaving. Otherwise, the punishment from having behaved badly becomes void.

### 4.1 Decentralized reputation mechanism

The decentralized mechanism is a combination of Sporas and CR, where each agent keeps the references given from other agents and calculates the reputation value, according to the formula:

$$R_{i+1} = \frac{1}{\theta}\sum_{1}^{t} \Phi(R_i) R_{i+1}^{other}(W_{i+1} - E(W_{i+1})) \quad \textbf{(1)}$$

$$\Phi(R) = 1 - \frac{1}{1 + e^{\frac{-(R-D)}{\sigma}}} \quad \text{and} \quad E(W_{i+1}) = \frac{R_t}{D}$$

where: $t$ is the number of ratings the user has received thus far, $\theta$ is a constant integer greater than 1, $W_i$ represents the rating given by user $i$, $R^{other}$ is the reputation value of the user giving the rating, $D$ is the range of reputation values (maximum rating minus minimum rating) and $\sigma$ is the acceleration factor of the damping function $\Phi$ (the smaller the value of s, the steeper the dumping factor $\Phi$). Note that the value of $\theta$ determines how fast the reputation value of the user changes after each rating. The larger the value of $\theta$, the longer the memory of the system is.

The user's rating value $W_i$ is based on four coefficients:

- *Correctness* ($Corr_i$): refers to the correctness of the returned results.
- *Completeness* ($Comp_i$): refers to the completeness of the returned results.
- *Response time* ($Resp_i$): refers to the Reasoner's response time.
- *Flexibility* ($Flex_i$): refers to the Reasoner's flexibility in input parameters.

The four coefficients are evaluated, based on the user's (subjective) assessment for each standard and their ratings vary from 1 to 10. The final rating value ($W_i$) is the weighted sum of the coefficients (equation (2) below), where $a_{i1}$, $a_{i2}$, $a_{i3}$ and $a_{i4}$ are the respective weights and $nCorr_i$, $nComp_i$, $nResp_i$ and $nFlex_i$ are the normalized values for correctness, completeness, response time and flexibility, accordingly:

$$w_i = a_{i1}nCorr_i + a_{i2}nComp_i + a_{i3}nResp_i + a_{i4}nFlex_i \quad \textbf{(2)}$$

New users start with a reputation equal to 0 and can advance up to the maximum of 3000. The reputation ratings vary from 0.1 for "terrible" to 1 for "perfect". Thus, as soon as the interaction ends, the Reasoner asks for a rating. The other agent responds with a new message containing both its rating and its personal reputation and the Reasoner applies equation (1) above to update its reputation.

### 4.2 Centralized reputation mechanism

In the centralized approach, a third-party agent keeps the references given from agents interacting with Reasoners or any other agent in the MAS environment. Each reference is in the form of:

$$Ref_i=(a, b, cr, cm, flx, rs)$$

where: $a$ is the *truster agent*, $b$ is the *trustee agent* and *cr* (*Correctness*), *cm* (*Completeness*), *flx* (*Flexibility*) and *rs* (*Response time*) are the evaluation criteria.

Ratings ($r$) vary from -1 (*terrible*) to 1 (*perfect*), while newcomers start with a reputation equal to 0 (*neutral*). The final reputation value ($R_b$) is based on the weighted sum of the relevant references stored in the third-party agent and is calculated according to the formula:

$$\sum R_b = w_1*cr + w_2*cm + w_3*flx + w_4*rs$$

where: $w_1+w_2+w_3+w_4=1$. Two options are supported for $R_b$, a default where the weights are equivalent, namely $w_{k\ s[1,4]}=0.25$ each and a user-defined, where the weights vary from 0 to 1 depending on user priorities.

### 4.3 Comparison

The simple evaluation formula of the centralized approach, compared to the decentralized one, leads to time gain as it needs less calculation time. Moreover, it provides more guaranteed and reliable results ($R_b$), as it is centralized, overcoming the difficulty to locate references in a distributed mechanism.

In addition, in the decentralized approach an agent can interact with only one agent per time and, thus, requires more interactions, in order to discover the most reliable agent, leading to further time loss.

Agents can use either of the above mechanisms or even both complementarily. Namely, they can use the centralized mechanism, in order to find the most trusted service provider and/or they can use the decentralized approach for the rest of the agents.

## 5 Use case: a brokering scenario

Defeasible reasoning (see section 3) is useful in various applications, like brokering [21], bargaining and agent negotiations [22]. These domains are also extensively influenced by agent-based technology [23]. Towards this direction, a defeasible reasoning-based brokering scenario is adopted from [24]. In order to demonstrate the functionality of the presented technologies, part of the above scenario is extended with deductive reasoning. Four independent parties are involved, represented by intercommunicating intelligent agents.

- The *customer* (called Carlo) is a potential renter that wishes to rent an apartment based on his requirements (e.g. location, floor) and preferences.
- The *broker* possesses a number of available apartments stored in a database. His role is to match Carlo's requirements with the features of the available apartments and eventually propose suitable flats to the potential renter.
- Two *Reasoners* (independent third-party services), *DR-Reasoner* and *R-Reasoner*, with a high reputation rating that can conduct inference on defeasible and

deductive logic rule bases, accordingly, and produce the results as an RDF file.

## 5.1 Scenario overview

The scenario is carried out in eight distinct steps, as shown in Fig. 4 Carlo's agent retrieves the corresponding apartment schema (Appendix A), published in the broker's website, formulates his requirements accordingly and submits them to the broker, in order to get back all the available apartments with the proper specifications (Fig. 4 – step 1). These requirements are expressed in defeasible logic, in the DR-DEVICE RuleML-like syntax (Fig 5 and Fig 6). For the interested reader, Appendix B features a full description of the customer's requirements in d-POSL (see Appendix E), a POSL[25]-like dialect for representing defeasible logic rule sets in a more compact way.

The broker, on the other hand, has a list of all available apartments, along with their specifications (stored as an RDF database – see **Figure** 7 for an excerpt), but does not reveal it to Carlo, because it's one of his most valuable assets. However, since the broker cannot process Carlo's requirements using defeasible logic, he requests a trusted third-party reasoning service. The DR-Reasoner, as mentioned, is an agent-based service that uses DR-DEVICE, in order to infer conclusions from a defeasible logic program and a set of facts in an RDF document. Hence, the broker sends the customer's requirements, along with the URI of the RDF document containing the list of available apartments, and stands by for the list of proper apartments (step 2).

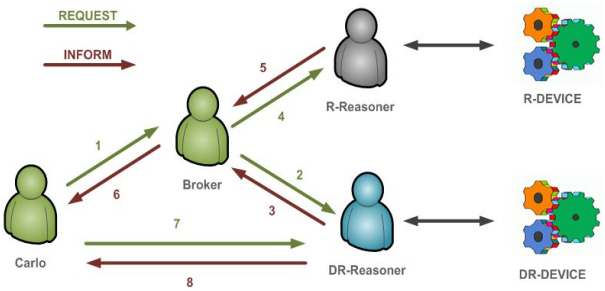


Figure 4: The distinct steps featured in the scenario.

Then, DR-Reasoner launches DR-DEVICE, which processes the above data and returns an RDF document, containing the apartments that fulfil all requirements (Fig. 8). When the result is ready, the Reasoner sends it back to the broker's agent (step 3). The latter should forward the results to Carlo's agent; however, the broker possesses a private "agenda", i.e. a rulebase that infers broker's proposals, according to his/her own strategy, customized to Carlo's case, i.e. selected from the list of apartments compatible to Carlo's requirements. A sample of these rules is shown in Appendix C; one rule proposes the biggest apartment in the city centre, while the other one suggests the apartment with the largest garden in the suburbs. These rules are formulated using deductive logic, so the broker sends them, along with the results of the previous inference step, to the R-Reasoner that launches R-DEVICE (step 4). Finally, the broker gets the appropriate list with proposed apartments that fulfil his "special" rules (step 5).

```
<RuleML rdf_import="...carlo_ex.rdf" rdf_export="export-carlo.rdf" >
.....
<Implies ruletype = "defeasiblerule">
 <oid><Ind uri = "&carlo_rb;r1">r1</Ind></oid>
 <head>
  <Atom>
   <op><Rel>acceptable</Rel></op>
   <slot><Ind>apartment</Ind><Var>x</Var></slot>
  </Atom>
 </head>
 <body>
  <Atom><op><Rel uri = "carlo:apartment"/></op>
   <slot><Ind>carlo:name</Ind><Var>x</Var></slot>
  </Atom>
 </body>
</Implies>
.....
</rulebase>
```

Figure 5: Rule base fragment – rule $r_1$.

```
<RuleML rdf_import="...carlo_ex.rdf" rdf_export="export-carlo.rdf" >
.....
<Implies ruletype = "defeasiblerule">
 <oid><Ind uri = "&carlo_rb;r2">r2</Ind></oid>
 <head>
  <Neg>
   <Atom>
    <op><Rel>acceptable</Rel></op>
    <slot><Ind>apartment</Ind><Var>x</Var></slot>
   </Atom>
  </Neg>
 </head>
 <body>
  <Atom>
   <op><Rel uri = "carlo:apartment"/></op>
   <slot><Ind>carlo:name</Ind><Var>x</Var></slot>
   <slot><Ind>carlo:bedrooms</Ind>
    <Constraint>
     <and_constraint><Var>y</Var>
      <Function_call name = "&lt;">
       <Var>y</Var><Ind>2</Ind>
      </Function_call>
     </and_constraint>
    </Constraint>
   </slot>
  </Atom>
 </body>
 <superior><Ind uri = "&carlo_rb;r1"/></superior>
</Implies>
.....
</rulebase>
```

Figure 6: Rule base fragment – rule $r_2$.

Eventually, Carlo receives the appropriate list (step 6) and has to decide which apartment he prefers. However, his agent does not want to send Carlo's preferences to the broker, because he is afraid that the broker might take advantage of that and will not present him with his most preferred choices. Thus, Carlo's agent sends the list of acceptable apartments (an RDF document) and his preferences (once again as a defeasible logic rule base) to the Reasoner (step 7). The latter calls DR-DEVICE and

gets the single most appropriate apartment. It replies to Carlo and proposes the best transaction (step 8). The procedure ends and Carlo can safely make the best choice based on his requirements and personal preferences. See Appendix D for a d-POSL version of Carlo's specific preferences. Notice that Carlo takes into consideration not only his preferences and requirements, but also broker's proposals, as long as they are compatible with his own requirements.

```
<rdf:RDF... xmlns:carlo="&carlo;">
  <carlo:apartment rdf:about="&carlo_ex;a1">
    <carlo:bedrooms rdf:datatype="&xsd;integer">1</carlo:bedrooms>
    <carlo:central>yes</carlo:central>
.....
  </carlo:apartment>
.....
</rdf:RDF>
```

Figure 7: RDF document excerpt for available apartments.

As for the reputation rating, after each interaction with the Reasoners, both the Broker and the Customer are requested for their ratings. For instance, after the successful end of step 3, the Broker not only proceeds to step 4, but also sends its rating to the Reasoner or/and the third-party agent. As a result, the latter updates the reputation value.

```
<!DOCTYPE rdf:RDF [
<!ENTITY dr-device "http://.../dr-device/export/export-carlo.rdf#"> ]>
<rdf:RDF... xmlns:dr-device='&dr-device;'>
.....
<dr-device:acceptable rdf:about="&dr-device;acceptable5">
<dr-device:apartment>a5</dr-device:apartment>
<dr-device:truthStatus>defeasibly-proven</dr-device:truthStatus>
</dr-device:acceptable >
.....
</rdf:RDF>
```

Figure 8: Results of defeasible reasoning exported as an RDF document.

## 5.2 Brokering protocol

Although FIPA provides standardized protocols, we found that none is suitable for our brokering scenario, since 1-1 automated brokering cannot be supported. As a result, a brokering protocol was implemented that encodes the allowed sequences of actions for the automation of the brokering process among the agents. The protocol is depicted in Fig. 9 and is based on specific performatives that conform to the FIPA ACL specification.

$S_0$ to $S_6$ represent the states of a brokering trade and $E$ is the final state. Predicates *Send* and *Receive* represent the interactions that cause state transitions. For instance, the sequence of transitions for the customer is: $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow E$, which means that the agent initially sends a *REQUEST* message ($S_1 \rightarrow S_2$) to the broker, then waits and finally gets an *INFORM* message with the response ($S_2 \rightarrow S_3$). After that, the customer decides to send a new request message to the DR-Reasoner ($S_3 \rightarrow S_4$), receives an *INFORM* message from him ($S_4 \rightarrow S_5$) and successfully terminates the process ($S_5 \rightarrow E$).

On the other hand, the transition sequence for the broker is: $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow E$. Initially, the agent is waiting for new requests; as soon as one is received ($S_0 \rightarrow S_1$), he sends an enriched *REQUEST* message to the DR-Reasoner ($S_1 \rightarrow S_2$) and waits for results. Finally, he gets the *INFORM* message from the DR-Reasoner ($S_2 \rightarrow S_3$) and sends a new enriched *REQUEST* message to the R-Reasoner ($S_3 \rightarrow S_4$). Eventually, the broker receives the appropriate *INFORM* message from the R-Reasoner ($S_4 \rightarrow S_5$) and forwards it to the customer ($S_5 \rightarrow S_6$), terminating the trade ($S_6 \rightarrow E$).

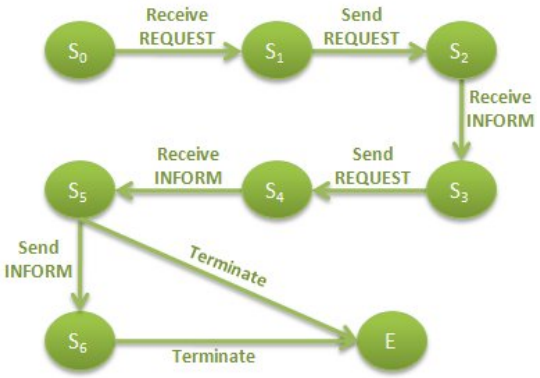


Figure 9: Agent brokering communication protocol.

In case that an agent receives a wrong performative, it sends back a NOT-UNDERSTOOD message and the interaction is repeated.

# 6 Related work

A similar architecture for intelligent agents is presented in [26], where various reasoning engines are employed as plug-in components, while agents intercommunicate via FIPA-based communication protocols. The framework is build on top of the OPAL agent platform [27] and, similarly to EMERALD, features distinct types of reasoning services that are implemented as reasoner agents. The featured reasoning engines are *3APL* [28], *JPRS* (*Java Procedural Reasoning System*) and *ROK* (*Rule-driven Object-oriented Knowledge-based System*) [29]. 3APL agents incorporate BDI logic elements and first-order logic features, providing constructs for implementing agent beliefs, declarative goals, basic capabilities and reasoning rules, through which an agent's goals can be updated or revised. JPRS agents perform goal-driven procedural reasoning and each JPRS agent is composed of a world model (agent beliefs), a plan library (plans that the agent can use to achieve its goals), a plan executor (reasoning module) and a set of goals. Finally, ROC agents are composed of a working memory, a rule-base (consisting of first-order, forward-chaining production rules) and a conflict set. Thus, following a similar approach to EMERALD, the framework integrates the three reasoning engines into OPAL in the form of OPAL micro-agents.

The primary difference between the two frameworks lies in the variety of reasoning services offered by EMERALD. While the three reasoners featured in [26] are all based on declarative rule languages, EMERALD proposes a variety of reasoning services, including deductive, defeasible and modal defeasible reasoning, thus, comprising a more integrated solution. Furthermore, the framework does not feature a trust and reputation mechanism. Finally, and most importantly, the approach of [26] is not based on Semantic Web standards, like EMERALD, for rule and data interchange.

The *Rule Responder* [30] project builds a service-oriented methodology and a rule-based middleware for interchanging rules in virtual organizations, as well as negotiating about their meaning. Rule Responder demonstrates the interoperation of various distributed platform-specific rule execution environments, based on Reaction RuleML as a platform-independent rule interchange format. We have a similar view of reasoning service for intelligent agents and usage of RuleML. Also, both approaches allow utilizing a variety of rule engines. However, contrary to Rule Responder, our framework (EMERALD) is based on FIPA specifications, achieving a fully FIPA-compliant model and proposes two reputation mechanisms to deal with trust issues. Finally, and most importantly, our framework does not rely on a single rule interchange language, but allows each agent to follow its own rule formalism, but still be able to exchange its rule base with other agents, which will use trusted third-party reasoning services to infer knowledge based on the received ruleset.

*DR-BROKERING*, a system for brokering and matchmaking, is presented in [31]. The system applies RDF in representing offerings and a deductive logical language for expressing requirements and preferences. Three agent types are featured (Buyer, Seller and Broker). Similarly, our approach identifies roles such as Broker and Buyer. On the other hand, we provide a number of independent reasoning services, offering both deductive and defeasible logic. Moreover, our approach takes into account trust issues, providing two reputation approaches in order to guarantee the interactions' safety.

In [32] a negotiation protocol and a framework that applies it are described. Similarly to our approach, the proposed framework also uses JADE. Additionally, a taxonomy of declarative rules for capturing a wide variety of negotiation mechanisms in a well-structured way is derived. The approach offers the same advantages with EMERALD, namely, the involved mechanisms are being represented in a more modular and explicit way. This makes agent design and implementation easier, reducing the risks of unintentional incorrect behaviour. On the other hand, EMERALD comprises a more generic framework, allowing the adoption of various scenarios that are not only restricted in negotiations. Moreover, reasoning services are provided, along with two reputation models for agents.

# 7 Conclusions

The paper argued that agent technology will play a vital role in the realization of the Semantic Web vision and presented a variety of reasoning services, 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. Additionally, since agents need a mechanism for establishing trust towards the reasoning services, reputation mechanisms (centralized and decentralized) were integrated in the framework and were also described in this work. Finally, the paper presents a use case brokering trade scenario that illustrates the usability of the technologies described in the paper.

As for future directions, it would be interesting to verify our model's capability to adapt to a variety of different scenarios other than brokering. An appealing field could be contract negotiation; the incorporation of negotiation elements into the agents' behavior would demand alterations in the protocol. The latter would now have to include the agents' negotiation strategy as well. Another goal is to integrate an even broader variety of distinct reasoning engines, thus, forming a flexible, generic environment for interoperating agents in the SW. Finally, our intention is to test our reasoning services (reasoners) in data intensive applications.

# References

[1] Berners-Lee T, Hendler J, Lassila O (2001) The Semantic Web. *Scientific American*, 284(5):34-43

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

[3] Boley H, Kifer M. *RIF Basic Logic Dialect*. Latest version available at http://www.w3.org/TR/rif-bld/.

[4] Nute D. (1987) Defeasible Reasoning. *20th International Conference on Systems Science*, IEEE Press, pp. 470-477.

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

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

[7] Abiteboul S, Hull R, Vianu V (1995) *Foundations of Databases*. Addison-Wesley, p. 305.

[8] Kifer M, Lausen G, Wu J (1995) Logical foundations of object-oriented and frame-based languages. *J. ACM* 42(4):741-843.

[9] Kozlenkov A, Penaloza R, Nigam V, Royer L, Dawelbait G, Schroeder M (2006) Prova: Rule-based Java Scripting for Distributed Web Applications: A Case Study in Bioinformatics. In Sebastian Schaffert (Ed.) *Workshop on Re-*

*activity on the Web at the International Conference on Extending Database Technology (EDBT 2006)*, Springer.

[10] Dietrich J, Kozlenkov A, Schroeder M, Wagner G (2003) Rule-based agents for the semantic web. *Electronic Commerce Research and Applications*, 2(4):323–338.

[11] Bassiliades N, Antoniou G, Vlahavas I (2006) A Defeasible Logic Reasoner for the Semantic Web. *International Journal on Semantic Web and Information Systems*, 2(1):1-41.

[12] Kravari K, Kontopoulos E, Bassiliades N (2009) Towards a Knowledge-based Framework for Agents Interacting in the Semantic Web. *2009 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'09)*, Italy, Vol. 2, pp. 482-485.

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

[14] Governatori, G, Rotolo, A (2008). BIO logical agents: Norms, beliefs, intentions in defeasible logic. *Journal of Autonomous Agents and Multi Agent Systems* 17:36–69.

[15] Berners-Lee T (1999) Weaving the Web, Harper San Francisco, ISBN: 0062515861.

[16] Berners-Lee T, Hall W, Hendler J, O'Hara K, Shadbolt N, Weitzner D (2006) A Framework for Web Science. *Foundations and Trends in Web Science*, Vol 1, No 1.

[17] Dasgupta P (2000) Trust as a commodity. Gambetta D. (Ed.). *Trust: Making and Breaking Cooperative Relations*, Blackwell, pp. 49-72.

[18] Macarthur K (2008) Tutorial: Trust and Reputation in Multi-Agent Systems. *International Conference on Autonomous Agents and Multiagent Systems* (*AAMAS 2008*), Portugal.

[19] Zacharia G, Moukas A, Maes P (2000) Collaborative reputation mechanisms for electronic marketplaces. *Decision Support Systems*, 29:371-388.

[20] Huynh T, Jennings N, Shadbolt N (2006) Certified Reputation: how an agent can trust a stranger. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, Hokkaido, Japan.

[21] Benjamins R, Wielinga B, Wielemaker J, Fensel D (1999) An Intelligent Agent for Brokering Problem-Solving Knowledge. *International Work-Conference on Artificial Neural Networks IWANN* (2), pp. 693-705.

[22] Governatori G, Dumas M, Hofstede A ter, Oaks P (2001) A Formal Approach to Protocols and Strategies for (Legal) Negotiation. *International Conference on Artificial Intelligence and Law (ICAIL 2001)*, pp. 168-177.

[23] Skylogiannis T, Antoniou G, Bassiliades N, Governatori G, Bikakis A (2007) DR-NEGOTIATE - A System for Automated Agent Negotiation with Defeasible Logic-based Strategies. Data & Knowledge Engineering (*DKE)*, 63(2):362-380.

[24] Antoniou G, Harmelen F van (2004) *A Semantic Web Primer*. MIT Press.

[25] Boley H.: POSL: An Integrated Positional-Slotted Language for Semantic Web Knowledge. http://www.ruleml.org/submission/ruleml-shortation.html

[26] Wang M, Purvis M, Nowostawski M. (2005) An Internal Agent Architecture Incorporating Standard Reasoning Components and Standards-based Agent Communication. In: *IEEE/WIC/ACM international Conference on intelligent Agent Technology (IAT'05)*, IEEE Computer Society, Washington, DC, pp. 58-64.

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

[28] Dastani M, van Riemsdijk M B, Meyer J-J C. (2005) Programming multi-agent systems in 3APL. In: R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni (Eds.) *Multi-Agent Programming: Languages, Platforms and Applications*, Springer, Berlin.

[29] Nowostawski, M. (2001) Kea Enterprise Agents Documentation.

[30] Paschke A, Boley H, Kozlenkov A, Craig B (2007) Rule responder: RuleML-based Agents for Distributed Collaboration on the Pragmatic Web. *2nd International Conference on Pragmatic Web*. ACM, pp. 17-28, vol. 280, Tilburg, The Netherlands.

[31] Antoniou G, Skylogiannis T, Bikakis A, Bassiliades N (2005) DR-BROKERING – A Defeasible Logic-Based System for Semantic Brokering. *IEEE International Conference on E-Technology, E-Commerce and E-Service*, IEEE, pp. 414-417.

[32] Bartolini C, Preist C, Jennings N (2002) A Generic Software Framework for Automated Negotiation. *1st International Joint Conference on the Autonomous Agents and Multi-Agent Systems (AAMAS)*, Italy.

# Appendix A – Apartment Schema

The RDF Schema file for the broker's apartments and proposals (Section 5):

```
<!DOCTYPE rdf:RDF [ <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <!ENTITY carlo "http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo.rdf#">
   <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#">
   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#"> ]>
<rdf:RDF xmlns:rdf="&rdf;" xmlns:carlo="&carlo;"
  xmlns:rdfs="&rdfs;" xmlns:xsd="&xsd;">
<rdfs:Class rdf:about="&carlo;apartment" rdfs:label="apartment">
   <rdfs:subClassOf rdf:resource="&rdfs;Resource"/>
</rdfs:Class>
<rdf:Property rdf:about="&carlo;bedrooms" rdfs:label="bedrooms">
   <rdfs:domain rdf:resource="&carlo;apartment"/>
   <rdfs:range rdf:resource="&xsd;integer"/>
</rdf:Property>
<rdf:Property rdf:about="&carlo;central" rdfs:label="central">
   <rdfs:domain rdf:resource="&carlo;apartment"/>
   <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdf:Property>
<rdf:Property rdf:about="&carlo;floor" rdfs:label="floor">
   <rdfs:domain rdf:resource="&carlo;apartment"/>
   <rdfs:range rdf:resource="&xsd;integer"/>
</rdf:Property>
<rdf:Property rdf:about="&carlo;gardenSize" rdfs:label="gardenSize">
   <rdfs:domain rdf:resource="&carlo;apartment"/>
   <rdfs:range rdf:resource="&xsd;integer"/>
</rdf:Property>
<rdf:Property rdf:about="&carlo;lift" rdfs:label="lift">
   <rdfs:domain rdf:resource="&carlo;apartment"/>
   <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdf:Property>
<rdf:Property rdf:about="&carlo;name" rdfs:label="name">
   <rdfs:domain rdf:resource="&carlo;apartment"/>
   <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdf:Property>
<rdf:Property rdf:about="&carlo;pets" rdfs:label="pets">
   <rdfs:domain rdf:resource="&carlo;apartment"/>
   <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdf:Property>
<rdf:Property rdf:about="&carlo;price" rdfs:label="price">
   <rdfs:domain rdf:resource="&carlo;apartment"/>
   <rdfs:range rdf:resource="&xsd;integer"/>
</rdf:Property>
<rdf:Property rdf:about="&carlo;size" rdfs:label="size">
   <rdfs:domain rdf:resource="&carlo;apartment"/>
   <rdfs:range rdf:resource="&xsd;integer"/>
</rdf:Property>
<rdfs:Class rdf:about="propose">
   <rdfs:subClassOf rdf:resource="&rdfs;Resource"/>
</rdfs:Class>
<rdf:Property rdf:about="apartment">
   <rdfs:domain rdf:resource="propose"/>
   <rdfs:range rdf:resource="&rdfs;Literal"/>
</rdf:Property>
</rdf:RDF>
```

# Appendix B – Carlo's Requirements

Carlo's requirements (Section 5) in d-POSL:

```
r1: acceptable(apartment->?x) :-
    apartment(name->?x).
r2: ~acceptable(apartment->?x) :-
    apartment(name->?x,bedrooms->?y), ?y > 2.
r3: ~acceptable(apartment->?x) :-
    apartment(name->?x,size->?y), ?y < 45.
r4: ~acceptable(apartment->?x) :-
    apartment(name->?x,pets->"no").
r5: ~acceptable(apartment->?x) :-
    apartment(name->?x,lift->"no",floor->?y), ?y > 2.
r6: ~acceptable(apartment->?x) :-
    apartment(name->?x,price->?y), ?y > 400.
r9: ~acceptable(apartment->?x) :-
    offer(apartment->?x,amount->?y),
    apartment(name->?x,price->?z), ?y < ?z.
r7: offer(apartment->?x,amount->?a) :-
    apartment(name->?x,size->?y,gardenSize->?z,central->"yes"),
    ?a is 300+2*?z+5*(?y-45).
r8: offer(apartment->?x,amount->?a) :-
    apartment(name->?x,size->?y,gardenSize->?z,central->"no"),
    ?a is 250+2*?z+5*(?y-45).
info-copy: apartment-info(apartment->?x,price->?p,
           size->?s,gardenSize->?gs) :-
               acceptable(apartment->?x),
               apartment(name->?x,price->?p,
               size->?s,gardenSize->?gs).
r2 > r1.
r3 > r1.
r4 > r1.
r5 > r1.
r6 > r1.
r9 > r1.
```

Rules $r_1$-$r_6$ express Carlo's requirements regarding the apartment specifications. Rules $r_7$ and $r_8$ indicate the offer Carlo is willing to make for an apartment that fits his needs, while rule $r_9$ ensures that the amount offered by the customer will not be higher than the apartment's actual rental price. Finally, rule `info-copy` stores all the characteristics of appropriate apartments that are of interest to Carlo, so that he can later refer to them.

# Appendix C – Broker's "Hidden Agenda"

Broker's "hidden agenda" (Section 5) in d-POSL:

```
propose(apartment->?x) :-
        acceptable(apartment->?x),
        apartment(name->?x,central->"yes",size->?s),
        \+ (acceptable(apartment->?y), ?x \= ?y,
        apartment(name->?y,central->"yes",size->?s1), ?s < ?s1).
propose(apartment->?x) :-
        acceptable(apartment->?x),
        apartment(name->?x,central->"no",gardenSize->?gs),
        \+ (acceptable(apartment->?y), ?x \= ?y,
        apartment(name->?y,central->"no",gardenSize->?gs1),
        ?gs < ?gs1).
```

The broker does not propose to Carlo all appropriate apartments, but only a subset of them, according to his "hidden agenda". The two rules depicted above are an example: the broker proposes to the customer the largest of all appropriate centrally located apartments or a non-centrally located one with the biggest garden size. Of course, the broker's hidden agenda could potentially consist of more (and possibly more adept) rules.

# Appendix D – Carlo's Preferences

Carlo's apartment preferences (Section 5) in d-POSL:

```
find_cheapest: cheapest(apartment->?x) :-
               acceptable(apartment->?x),
               apartment-info(apartment->?x,price->?z),
               \+ (acceptable(apartment->?y),
               apartment-info(apartment->?y,price->?w),
               ?x \= ?y, ?w < ?z).
find_largest: largest(apartment->?x) :-
              acceptable(apartment->?x),
              apartment-info(apartment->?x,size->?z),
              \+ (acceptable(apartment->?y),
              apartment-info(apartment->?y,size->?w),
              ?x \= ?y, ?w < ?z).
find_largestGarden: largestGarden(apartment->?x) :-
                    acceptable(apartment->?x),
                    apartment-info(apartment->?x,gardenSize->?z),
                    \+ (acceptable(apartment->?y),
                    apartment-info(apartment->?y,gardenSize->?w),
                    ?x \= ?y, ?w < ?z).
r10: rent(apartment->?x) :-
     propose(apartment->?x), cheapest(apartment->?x).
r11: rent(apartment->?x) :-
     propose(apartment->?x), cheapest(apartment->?x),
     largestGarden(apartment->?x).
r12: rent(apartment->?x) :-
     propose(apartment->?x), cheapest(apartment->?x),
     largestGarden(apartment->?x),largest(apartment->?x).
r11 > r10.
r12 > r10.
r12 > r11.
:- rent(apartment->?x), rent(apartment->?y), ?x \= ?y.
```

Carlo will choose among the apartments proposed by the broker and the ones that are compatible with his own preferences.

# Appendix E – d-POSL

*POSL* (positional-slotted language) [26] is an ASCII language that integrates Prolog's positional and F-logic's

slotted syntaxes for representing knowledge (facts and rules) in the Semantic Web. POSL is primarily designed for human consumption, since it is faster to write and easier to read than any XML-based syntax. We devised an extension to POSL, called *d-POSL*, which handles the specifics of defeasible logics and is a secondary contribution included in this work. Variables are denoted with a preceding "?". A deeper insight into core POSL, its unification scheme, the underlying webizing process (i.e. the introduction of URIs as names in a system to scale it to the Web – orthogonal to the positional/slotted distinction), and its typing conventions along with examples is found in [26].

Furthermore, d-POSL maintains all the critical components of POSL, extending the language with elements that are essential in defeasible logics:

- Rule Type: Binary infix functors are introduced (":-", ":=",":~") to denote the rule type ("strict", "defeasible", "defeater", respectively).
- Rule Label: The rule label is a vital feature in defeasible logic, since it satisfies the need to express superiorities among rules. Consequently, d-POSL employs a mechanism for expressing rule labels and superiority relationships.
- Conflicting Literals: Conflicting literals are represented as headless rules, i.e. constraints that have the following format:
  `:= predicate(?x), predicate(?y), ?x\=?y.`
  See, for example, Appendix D above.