Computer Science and Information Systems 0(0):1–6 DOI: N/A

Agent Reasoning on the Web using Web Services*

Costin Bădică[1], Nick Bassiliades[2], Sorin Ilie[1], and Kalliopi Kravari[2]

[1] University of Craiova
Bvd. Decebal, 107, RO-200440, Craiova, Romania
{cbadica,silie}@software.ucv.ro
[2] Aristotle University of Thessaloniki, GR-54124, Thessaloniki, Greece
{nbassili,kkravari}@csd.auth.gr

Abstract. In this paper we present an approach for reusing agent-based reasoning capabilities by making them available for invocation as Web services. In this way, we provide the missing link between the highly interoperable Web services and the autonomicity and intelligence of agent-based systems, so that the latter can be seamlessly integrated into the knowledge-rich Semantic Web environment without being compromised by isolated communication platforms and languages or restricted to only one or just few reasoning formalisms. We have achieved this by extending the EMERALD framework for agent based reasoning with a Web service interface. Our approach is exemplified by the development of an online system for intelligent brokering of apartment rentals. The broker intelligence is captured as a defeasible knowledge base, while its problem solving process involves the invocation of third party defeasible reasoning Web services included into the EMERALD framework.

Keywords: Web service, reasoning, software agent, UML.

1. Introduction

The Web was originally envisioned as a platform for information dissemination to human consumers. The next generation of Web systems shifted its interest to make that information available for machine consumption by realizing that Web data can be reused for various problem solving purposes. This idea was put into practice through the development of the Semantic Web [2] that promotes the enrichment of Web content with meaningful semantic metadata.

In parallel with the progress of the Semantic Web, the field of distributed systems has seen a progress by the proposal of a new approach known as service oriented architecture. By promoting the new ideas of interoperability and integration, the adoption of this approach enabled the development of significantly more complex distributed applications by integration of various existing or new heterogeneous components.

Rapidly, this idea was embraced by the Web community resulting in the adoption of the service oriented architecture as a method for developing more complex Web-based applications using the Web services concepts, standards and technologies. Naturally, the

* Acknowledgments: This work was supported by the *K-SWAN: An Interoperable Knowledge-based Framework for Negotiating Semantic Web Agents* Greek-Romanian bilateral research project carried out between 2012-2013 and partly funded by Romanian UEFISCDI and by the Greek R&D General Secretariat.

areas of the Semantic Web and Web services converged to the new concept of Semantic Web services. They can be broadly understood either as Web services whose description is enhanced with semantic metadata to allow automated discovery and composition, or as Web services capable of providing declarative semantics in the form of reasoning methods, beyond the operational semantics of procedural invocation.

Software agents are defined as computer systems situated in an environment that are able to achieve their objectives by: (i) acting autonomously, i.e. by deciding themselves what to do, and (ii) being sociable, i.e. by interacting with other software or human agents [33]. If we place software agents into the Web environment then we can easily regard them as a potential incarnation of the Semantic Web vision. According to this view, the Semantic Web is regarded as a Web of semantic services that are provided and consumed by software agents over the Web protocols.

We already have service provisioning agents that act similarly with Web services being able to provide reasoning services [18] or negotiation services [9]. However, we have noticed that there are both conceptual and technological "missing links" between software agents and Web services in the context of the Semantic Web. From the conceptual point of view, services lack autonomy and they are rather passive components that behave reactively by waiting to be invoked by client software. They also lack social abilities, as well as the ability to adapt and evolve. On the other hand agents have interesting features regarding autonomy, sociability and adaptivity. From the technological point of view, multi-agent systems are developed using specialized packages and deployed on distributed multi-agent platforms. Those platforms are providing appropriate middleware services required for running distributed multi-agent systems [7]. Moreover, while available for invocation within the agent platform, agent services are not immediately available for invocation over the Web. In this paper we propose a system that integrates and updates existing software tools to enable the invocation of agent reasoning services over the Web. In particular we highlight the problems, as well as our solution details of the integration process of a defeasible reasoning service into a coherent system for supporting a realistic e-commerce scenario in the domain of brokering apartment rental.

Our work contributes to bridging gap between software agents and Web services in the context of the Semantic Web. Using our approach, a Web application can reuse a set of declarative rules that capture valuable knowledge about the various constraints, requirements and particularities of the problem domain. The knowledge can be represented using one of the available rule-based knowledge representation formalisms. Using our extension, Web applications can exploit this knowledge by invoking appropriate reasoning services over the Web protocols on specific sets of facts that capture the particular instance of the problem in hand.

The main contributions reported in this paper can be summarized as follows:

(i) The extension of EMERALD multi-agent knowledge-based framework [17] to enable the exposure of agent-based reasoning services as Web services.
(ii) The development of a prototype system supporting a realistic scenario for intelligent brokering of apartment rentals. The broker intelligence is captured as a defeasible knowledge base, while its problem solving process involves the invocation of third party defeasible reasoning Web services included into the EMERALD framework. The prototype provides a Web-based user interface, incorporates a rule-based knowledge base about user preferences and invokes external Web services for reasoning.

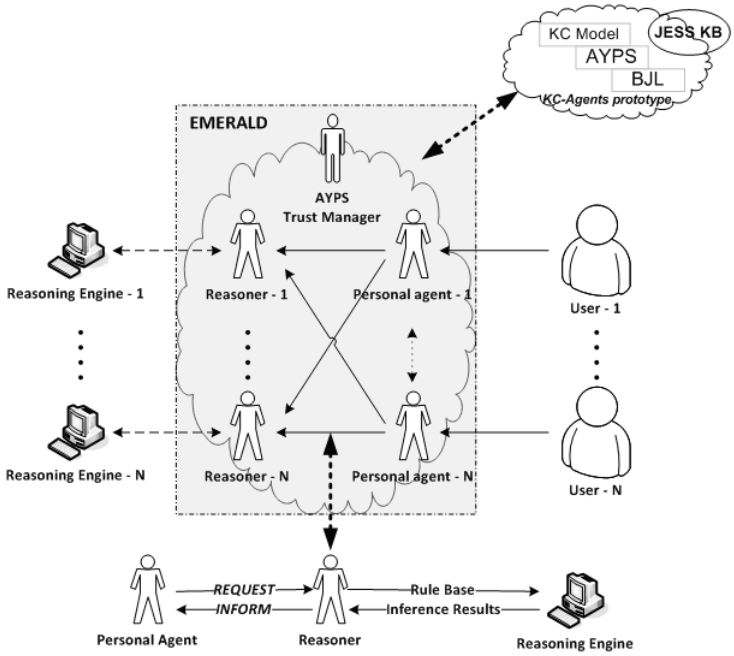


Fig. 1. EMERALD abstract architecture.

This prototype is intended as a proof-of-concept for checking the viability of our proposed approach.

Note that, although in our experimental system we used the particular *DR Reasoner* included into the EMERALD framework [16] and the scenario involves an e-commerce brokering activity for apartments rental, the presented approach is general and customizable so it can be also applied (i) to other types of reasoners, as well as (ii) to other application domains.

The paper is structured as follows. In section 2 we give an overview of background results and software tools that we have utilized to develop our prototype system and we also provide a review of related works. Section 3 introduces our method for exposing agent-based reasoners incorporated into EMERALD as Web services. The method was used for the development of the *DR Web Service* for defeasible reasoning. In section 4 we present our use case and we outline the design and implementation of an intelligent broker for the domain of apartment rental. Section 5 presents the details of our experiment carried out with the prototype system. The last section presents our conclusions and identifies a number of paths for continuing this research.

2. Background and Related Works

In this section we provide a brief overview of software platforms and components used in our proposed system, as well as an outlook to the results and related works from the literature.

2.1. Background

EMERALD ([17], see Fig. 1) is a multi-agent knowledge-based framework, based on the Semantic Web and FIPA standards [10], that enables reusability and interoperability of behavior between agents. It is built on Jade [4], a reliable and widely used framework. EMERALD supported so far the implementation of various applications, like brokering and agent negotiations. It provides a generic, reusable agent prototype for knowledge-customizable agents (KC-Agents), consisted of an agent model (KC Model), a directory service (AYPS - Advanced Yellow Pages Service) and external Java methods (Basic Java Library). Hence, EMERALD supports service provisioning by allowing agent service descriptions to be published onto the "yellow pages" service - known as Directory Facilitator (DF) in FIPA and Jade. Whenever a consumer agent decides to outsource a certain function, it can search the AYPS for an appropriate service that meets its specific requirements. AYPS actually provides a service retrieval ability, which groups and sorts the registered (advertised) services according, among others, to their domain and their synonyms, allowing (requesting) agents to make complex queries and receive the best available service. Yet since trust has been recognized as a key issue in Semantic Web Multi-agent Systems, EMERALD adopts a variety of reputation mechanisms, both decentralized and centralized [26].

Another important issue in this framework is reasoning interoperability. Agents do not necessarily share a common rule or logic formalism, thus, it is vital for them to find a way to exchange their position arguments seamlessly. To this end, EMERALD proposes the use of Reasoners [18], which are actually agents that offer reasoning services to the rest of the agent community. This approach does not rely on translation between rule formalisms, but 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 rule base, namely the set of entailments of the knowledge base. Thus, although Reasoners are built as agents, actually they act more like Web services.

Currently, EMERALD implements a number of Reasoners that offer reasoning services in two major reasoning paradigms: deductive rules and defeasible logic. 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 [23], on the other hand, constitutes a non-monotonic rule-based approach for efficient reasoning with incomplete and inconsistent information, which is useful in many applications, such as security policies, business rules, e-contracting, personalization, brokering and agent negotiations.

Among these Reasoners available in EMERALD, DR-Reasoner based on DR-DEVICE [3] has an exceptional interest. It is based on defeasible logic (DL), a nonmonotonic logic which is capable of modeling the way intelligent agents, like humans, draw reasonable conclusions from inconclusive information. Knowledge in DL is represented in the form of facts and rules. Facts are indisputable statements, represented either in form

of states of affairs or actions that have been performed. On the other hand, rules describe the relationship between premises and conclusion. Three types of rules are available strict rules, defeasible rules and defeaters. Strict rules ($A_1, \ldots, A_n \rightarrow B$) are rules in the classical sense: whenever the premises are indisputable then so is the conclusion. Thus, they can be used for definitional clauses. Defeasible rules ($A_1, \ldots, A_n \Rightarrow B$) are rules that can be defeated by contrary evidence. Defeaters ($A \rightsquigarrow p$), finally, are used to prevent conclusions, not to support them which is actually the main concept in DL. This logic does not support contradictory conclusions, instead seeks to resolve conflicts.

Hence, in cases where there is some support for concluding A but there is also support for concluding $\neg A$, namely where there are conflicting (mutually exclusive) literals, no conclusion can be derived unless one of them has priority over the other. This priority is expressed through a superiority relation among rules which defines priorities among them; namely where one rule may override the conclusion of another rule. For example, suppose that there is a conflict set $\{(c(x), c(d)|x \neq d\}$; it is consisted of two competing rules $r1$ ($r1 : a(X) \Rightarrow c(X)$) and $r2$ ($r2 : b(X) \Rightarrow c(X)$), where $r2$ could be superior ($r2 > r1$). To this end, ***DR Reasoner*** in order to exploit the DL's capabilities uses, as already mentioned, the DR-DEVICE inference system. DR-DEVICE is capable of reasoning about RDF metadata over multiple Web sources using defeasible logic rules. More specifically, DR-DEVICE accepts as input the address of a defeasible logic rule base, written in DR-RuleML language, an extension of the OORuleML syntax. The rule base contains the rules and one or more RDF documents that contain the facts for the rule program, while the final 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. DR-DEVICE is based on a CLIPS-based implementation of deductive rules. The core of the system consists of a translation of defeasible knowledge into a set of deductive rules, including derived and aggregate attributes. The implementation is declarative as it interprets the not operator using Well-Founded Semantics [11].

An important aspect of our approach was the integration of multi-agent technologies with Web services. Authors of [25] provide a good overview of such possibilities with a special focus on interoperability between FIPA (Foundation for Intelligent Physical agents) and Web services. Among them, a useful tool for our own work is the Jade Web service Integration Gateway – WSIG (see [4], chapter 10). It allows the provision of Jade agent services as Web services. In particular, services provided by Reasoner agents available in EMERALD framework can be exposed as Web services. This allows the convenient incorporation of reasoning functions into complex Semantic Web applications.

2.2. Related Works

Concerning interoperability, Rule Responder [5] is quite similar to EMERALD. It builds a service-oriented methodology and a rule-based middleware for interchanging rules in virtual organizations. It demonstrates the interoperation of distributed platform-specific rule execution environments, with Reaction RuleML as a platform-independent rule interchange format. It has a similar view of reasoning service for agents and usage of RuleML but it is not based on FIPA specifications. EMERALD supports multiple rule formalisms via trusted third-party reasoning services rather than a single rule interchange language.

In EMERALD, Reasoners instead of interchanging rule bases using a specific rule language, they have a simpler, but more general interface, that requires just the URL or the file path of the rulebase, in the form of Java Strings. This information is exchanged via ACL messages either from a requesting agent to a rule engine or from the rule engine to the requesting agent. Hence, it is up to the requesting agent to provide the appropriate files, by taking each time into consideration the rule engines' specifications. For instance, an agent who wants to use the DR-DEVICE rule engine has to provide valid RuleML files. To this end, whenever a Reasoner receives a new valid request, it launches a new instance of the associated reasoning engine that performs inference. Reasoners actually act like proxies, allowing other agents to contact with the appropriate rule engines with low computational and time cost. Otherwise, each single agent should have had hard-coded rule engines for every rule or logic formalism, which is infeasible. Moreover, EMERALD provides trust mechanisms that ensure the reliability of each agent or Reasoner, in particular, acting in the environment. Hence, it is ensured that Reasoners in EMERALD are trusted parties that have no access or interference in agents' internal information.

A similar to EMERALD architecture for intelligent agents is presented in [32], where 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 [24] and, similarly to EMERALD, features distinct types of reasoning services that are implemented as reasoner agents. The featured reasoning engines are 3APL [8], JPRS (Java Procedural Reasoning System) and ROC (Rule-driven Object-oriented Knowledge-based System). 3APL agents incorporate BDI logic elements and first-order logic features, providing constructs for implementing agent beliefs, declarative goals, basic capabilities and reasoning rules. JPRS agents perform goal-driven procedural reasoning and each JPRS agent is composed of a world model (agent beliefs), a plan library, 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. The primary difference between the two frameworks lies in the variety of reasoning services offered by EMERALD. While the three reasoners featured in [8] 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, EMERALD not only features trust and reputation mechanisms but also it is based on the Semantic Web standards, like, for rule and data interchange.

The One Ring project [22] on the other hand is a promising scripting rules engine service. It aims to be used as a Web service (SOA) for multiple applications to gain access to scripted processing of arbitrary parameters. It centralizes processing of common or business rules for multiple applications that need access to the same rules. Rules are defined using a simple language understood by domain experts. The primary difference between One Ring and EMERALD lies in the variety of reasoning services offered by EMERALD. While One Ring supports only a simple but not widely understood language, EMERALD proposes a variety of reasoning services, including deductive, defeasible and modal defeasible reasoning. Yet EMERALD, featuring additionally trust mechanisms, comprises a more integrated solution complying with the Semantic Web standards. However, both approached share a similar view of rule importance and usage.

Our literature review revealed also a number of systems and approaches that combine distributed multi-agent systems, reasoning services and the Semantic Web technologies for the development of application in several areas including: medical rehabilitation, agent auctions, supply-chain management, and service personalization.

Reference [29] proposes an ontology-based medical rehabilitation system called OntoRis that provides rehabilitation-related expertise to patients and therapists. The system uses agent technology for exploring and integrating external knowledge resources into the application. Sharing of users experience is encouraged in OntoRis with the help of an Web 2.0-enabled discussion forum. To ensure a certain level of correctness of information, the users experience in the forum is filtered by evidence-based medicine, thus only certified information will be integrated into OntoRis. The OntoRis component incorporates an ontology and a reasoning engine. Nevertheless, while OntoRis agents are useful for information searching and knowledge enhancement, they have no role in the automation of reasoning processes, requiring either the manual intervention of the OntoRis administrator, as well as knowledge certification via methods of evidence-based medicine.

Knowledge-based declarative approaches are also useful for the specification of interaction mechanisms between software agents. Reference [21] proposes a declarative framework for auctions mechanisms in multi-agent systems. The key point of the proposal is the factoring of the model into two architectural components, the auction host and the auction participants. Their interaction is governed by a protocol with a declarative specification decomposed into: generic negotiation protocol (GNP), declarative negotiation mechanism (DNM) and custom negotiation mechanism (CNS). The GNP makes possible the interaction between auction host and auction participants. The specific rules of a particular auction are described using the DNM, while the CNS captures the strategic behavior which is private to each auction participant. A prototype implementation of this model is proposed using the Jason agent programming language [6]. This implementation benefitted from several features of Jason including: Belief-Desire-Intention model, Prolog-style rule-based reasoning, meta-programming and extensibility. Nevertheless, a cleaner separation between the CNS and DNM is desirable, especially taking into account that negotiation strategies could require more specialized types of reasonings, for example to account for priorities or conflicts between strategy rules.

Authors of reference [31] propose an ontology-based approach for capturing negotiation knowledge in supply chain management multi-agent systems. Similarly to [21], negotiation knowledge is partitioned into shared negotiation ontology and private negotiation ontology to ensure agent communicative interoperability and privacy of strategic knowledge. The agents' negotiation ability is further enhanced with private inference rules defined on top of private negotiation ontology. Nevertheless, although conceptually quite similar to [21], the implementation approach is different. It is based on Jade multi-agent platform and Jess expert system shell [12], as well as on the Semantic Web technologies OWL and SWRL [30].

Personalization is an important concern in context-aware applications. Authors of [27] propose a method that combines ontological modeling of user profiles, rule-based personalization mechanisms and service-oriented architecture for providing personalized Help-on-Demand services to mobile users in pervasive environments. The flexibility of the method is achieved by using an intelligent personalization service that incorporates a rule-based knowledge and a reasoning engine. While interesting, one limitation of this

approach is that it is restricted to a single type of reasoning based on standard forward chaining for determining cause-effect relationships.

Reference [20] proposes the use of agent-based Web services to enhance collaboration within a supply chain of retailers, manufacturers and suppliers. However, rather than looking at the possibilities for integrating agent and Web services technologies to enhance flexibility of reasoning, the paper is focused on defining and experimentally evaluating of two scenarios: (i) independence level, where collaboration is kept minimal by letting agents have individual goals and communicate minimally by peer-to-peer information exchange, and (ii) collaboration level, by letting partner agents develop strategic relationships through intense communication thus allowing to achieve global goals for the entire supply chain.

3. Agent-Based Reasoning Web Services

The aim of this section is to introduce the design and implementation of a Reasoning Web Service. An external Web-based application will be able to invoke this Web service by providing a rulebase and an initial set of facts. The Web service returns the output of the reasoning process to the invoker application as a set of resulted facts.

3.1. Architecture and Design

Our design presents a method for wrapping an EMERALD reasoning service as a Web service using the Jade WSIG. For that purpose we exploit:

- The architecture and interoperability protocols of Reasoner agents inside EMERALD framework.
- The architecture of Jade WSIG.

The Reasoning Web Service takes as input a rule base represented in RuleML and outputs a result file using a simple request-reply interaction protocol. The meaning of the rulesbase, as well as of the initial set of facts is application dependent. It must be appropriately defined and interpreted by the client application in the context of the application domain of the system that is using the reasoning process.

The rulebase will specify a link to an RDF file containing an initial set of facts using the *rdf_import* attribute of RuleML. The output of the reasoning process is determined as an RDF file and saved by the Reasoning Web Service in a temporary location on the service side, while its URL is returned to the client. Consequently, the client will be able to download the result file and further process it according to the application requirements.

The architecture of the Reasoning Web Service is shown in Fig. 2. In particular, this diagram shows the relation between EMERALD, Jade, WSIG and the "outside world". The Reasoning Web Service is hosted by a Web server and implemented with the help of a WSIG servlet that handles reasoning requests. The servlet is interfaced with Jade via a special agent – the WSIG agent using a special component known as the Jade Gateway. The WSIG agent is able to pass reasoning requests to the appropriate Reasoner agent of the EMERALD framework, as well as to return reasoning results to the WSIG servlet.

For the design of the Reasoning Web Service we have produced an UML activity diagram that shows the activities of the client application and the reasoning platform, shown

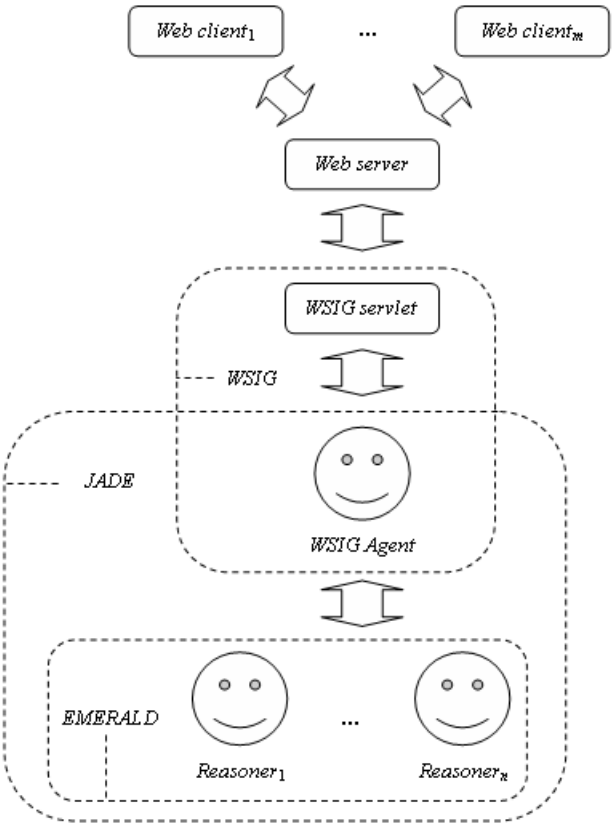


Fig. 2. Block diagram of Reasoning Web Service.

in Fig. 3, as well as an UML sequence diagram detailing the interactions between them, shown in Fig.4. For the reasoning platform we show two actors, namely the *Reasoning Web Service* which was developed by us, as well as the *Reasoner* which is part of EMERALD. The system functioning can be better understood if we follow both diagrams.

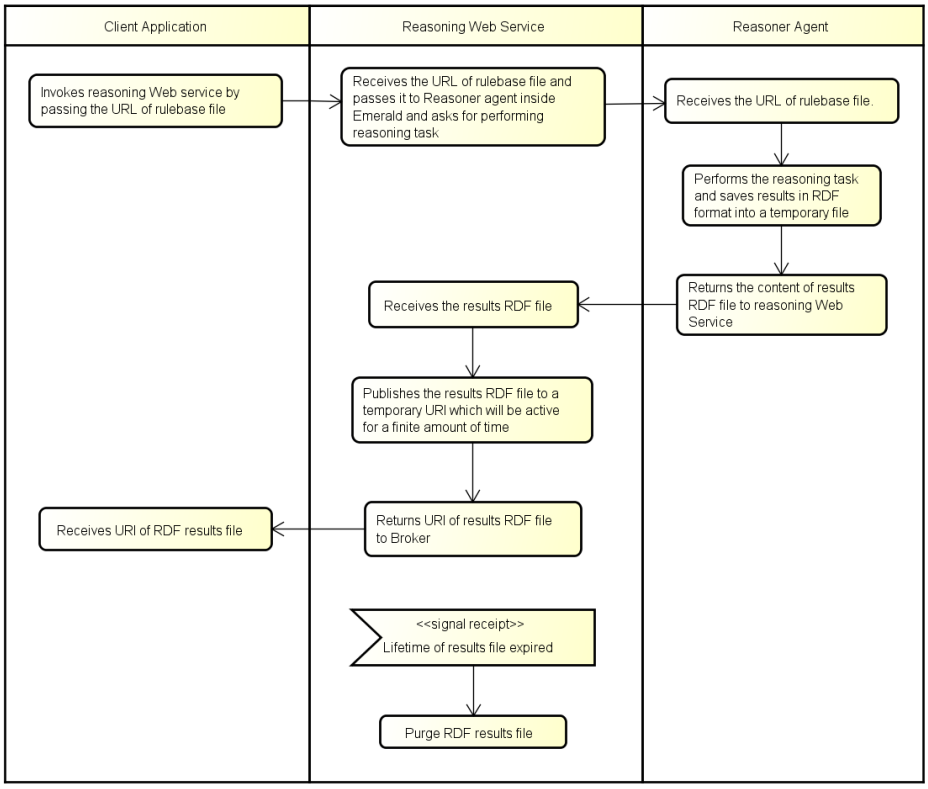


Fig. 3. Roles and activities in the Reasoning Web Service.

The *Client Application* invokes the *Reasoning Web Service* (interaction 1 in Fig.4) by passing it the *rulebase*. The *Reasoning Web Service* simply extracts the *rulebase* from the request and forwards it to the *Reasoner Agent* (interaction 2 in Fig.4). The result of the reasoning process expressed in RDF (*resultsRDF*) is returned to the *Reasoning Web Service* (interaction 3 in Fig.4). The *Reasoning Web Service* publishes the RDF file to a temporary URL and then returns this URL (*resultsURL*) to the *Client Application* (interaction 4 in Fig.4). The *Client Application* receives the URL and consequently can download the RDF file and further process it according to the application requirements. The temporary RDF file containing the reasoning results will reside on the Web server that hosts the Reasoning Web Service for a preestablished finite amount of time. When this time elapses the results file will be automatically purged, as can be seen in Fig.3.

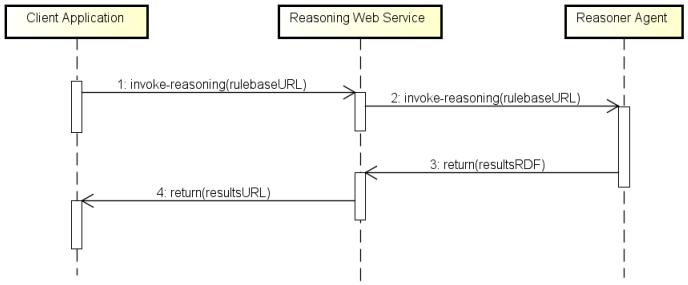


Fig. 4. Roles interaction in the Reasoning Web Service.

3.2. Implementation

For demonstration purposes we have only addressed the integration of the *DR Reasoner* agent available in EMERALD into our prototype system. Nevertheless, the principles behind this integration process are the same for any other Reasoner that is part of the EMERALD platform. As a general procedure, a Reasoner (including the *DR Reasoner*): (i) receives a rulebase that contains the knowledge base as a set of rules expressed for interoperability purposes using RuleML, as well as possibly one or more RDF files containing initial facts that are required for the reasoning process, (ii) performs the reasoning and (iii) returns the reasoning result as an RDF file.

For the implementation we have used the Jade WSIG add-on [13] that gives the possibility of invoking agent services exposed as Web services from Web service clients. WSIG provides the standard Web services stack including: WSDL service descriptions, SOAP message transport and UDDI "yellow pages" repository. WSIG runs as part of a Web application powered by a Web server and a servlet container. WSIG incorporates the WSIG servlet and the WSIG agent and acts as a gateway between the Web application and the Jade multi-agent system. The WSIG servlet handles SOAP over HTTP requests, extracts the SOAP content, maps it to an agent action and passes it to the WSIG agent. After the action execution, WSIG servlet converts the result received from the WSIG agent to SOAP, prepares a SOAP over HTTP reply and sends it back to the Web client (see Fig. 2). The implementation follows the architecture diagram shown in Fig. 2.

The WSIG add-on provides a method for exposing Jade agent services whose descriptions are published in the *Directory Facilitator* (DF) agent as Web services. For this purpose, the DF description of an agent service is automatically mapped by WSIG onto an WSDL description. For example, the WSDL for the reasoning service provided by *DR Reasoner* is shown in Fig. 5. Analyzing this example we observe that this WSDL file defines a Web service named *Defeasible_Reasoning_Service* that was derived from the description of the *Defeasible_Reasoning_Service* service provided by *DR Reasoner* agent.

The WSDL is structured in an abstract part, containing data types, messages and operations specification, followed by a concrete part, containing bindings and services specification [1]. The abstract part contains: (i) type definitions specified using XML Schema – the *wsdl:type* section; (ii) message types, specified as *wsdl:message* sections; and (iii) port types that represent service interfaces (expressed using *wsdl:portype* ele-

```
<wsdl:definitions ... declarations of namespaces>
  <wsdl:types>
    <xsd:schema xmlns:impl="urn:Defeasible_Reasoning_Service"
      ... declarations of namespaces>
      <xsd:annotation/>
      <xsd:element name="DRReasoning">
        <xsd:complexType>
          <xsd:sequence><xsd:element
            name="inputRulemlPath" type="xsd:string"/></xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="DRReasoningResponse">
        <xsd:complexType>
          <xsd:sequence><xsd:element
            name="DRReasoningReturn" type="xsd:string"/></xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="DRReasoningRequest">
    <wsdl:part name="parameters" element="impl:DRReasoning"></wsdl:part>
  </wsdl:message>
  <wsdl:message name="DRReasoningResponse">
    <wsdl:part name="parameters" element="impl:DRReasoningResponse"></wsdl:part>
  </wsdl:message>
  <wsdl:portType name="Defeasible_Reasoning_ServicePort">
    <wsdl:operation name="DRReasoning">
      <wsdl:input message="impl:DRReasoningRequest"></wsdl:input>
      <wsdl:output message="impl:DRReasoningResponse"></wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding
    name="Defeasible_Reasoning_ServiceBinding"
    type="impl:Defeasible_Reasoning_ServicePort">
    <wsdlsoap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="DRReasoning">
      <wsdlsoap:operation soapAction="urn:Defeasible_Reasoning_ServiceAction"/>
      <wsdl:input><wsdlsoap:body use="literal"/></wsdl:input>
      <wsdl:output><wsdlsoap:body use="literal"/></wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="Defeasible_Reasoning_ServiceService">
    <wsdl:port name="Defeasible_Reasoning_ServicePort"
      binding="impl:Defeasible_Reasoning_ServiceBinding">
      <wsdlsoap:address location=
        "http://ids.software.ucv.ro:8080/KSWAN/ws/Defeasible_Reasoning_Service"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Fig. 5. WSDL of the DR Web service.

ment) made up of sets of operations, represented as *wsdl:operation* elements. For example our *Defeasible_Reasoning_Service* Web service has an interface with a single operation that takes two parameters: an input parameter representing a *DRReasoningRequest* message and an output parameter representing a *DRReasoningResponse* message. Going deeper, for example the *DRReasoningRequest* message transports an object of type *DRReasoning* that encapsulates a value named *inputRulemlPath* representing the URL of the rulebase that is passed to the service. The concrete part contains bindings for operations and messages (element *wsdl:binding*) and service endpoints specification (element *wsdl:port*). For example, *Defeasible_Reasoning_Service* Web service exchanges

messages that are specified in "document" style and are formatted using SOAP and transported over HTTP. This binding is combined with the URL where this service is available: *http://ids.software.ucv.ro:8080/KSWAN/ws/Defeasible_Reasoning_Service*.

4. Web-Enabled Intelligent Broker

This section contains the details of the use case, as well as the design and implementation of our prototype Web-enabled Intelligent Broker that is using third party reasoning services provided by our Reasoning Web Service.

4.1. Use Case

We consider the intelligent broker use case introduced in [3] and we adapt it for the Web environment. The actors of this use case are:

(i) The *User* that represents a buyer interested to purchase or rent an apartment.
(ii) The *Broker* that matches buyer requested features with seller capabilities and recommends one or more possible transactions that meet the requirements of both trading parties.
(iii) The EMERALD framework that provides on-demand third party reasoning services for supporting the "intelligence" of the *Broker*.

Placing the actors in the Web environment, the situation presents as follows. The *User* is using a Web browser to connect to the *Broker* Web site and to search for apartment offers. The business of the *Broker* is powered by a Web application that provides at least the following two functionalities: (i) to assist the human users looking for renting apartments to search for suitable offers according to their requirements; (ii) to allow landlords to describe and publish their offers. Apartment offers will be registered and saved in a database that is available to the Web application of the *Broker*. In this work we assume the existence of this database and we shall only focus on the *User* side that is looking for renting or buying a suitable apartment.

The intelligence of the *Broker* is driven by a knowledge base. However, taking into account the large variety of knowledge representation formalisms, in particular based on rules, the *Broker* decided to outsource its reasoning functions to a third party reasoning platform (EMERALD in this case) that provides trusted reasoning capabilities that can be invoked over the Web as Web services. Separating the representation of the *Broker* intelligence from the reasoning services has the advantage that the *Broker* can update the knowledge-base according to the interests of the *User*. This update can either involve the update of the rules, as well as the adoption of a new rule representation formalism that better fits the *User* needs.

Briefly, the brokering use case proceeds as follows. In the first step the *User* connects to the *Broker* Web site and receives a form where he or she can fill-in his or her request. When the input process is finished, the *Broker* generates a parameterized rulebase represented using RuleML and an RDF file that contains parameter values specified by the *User*. Parameters can specify for example: minimum number of bedrooms, maximum floor, minim size of the garden (in m^2), if pets are allowed or not, maximum price for

a central or suburb location, extra price per m^2 (above the minimum preferred size), the minimum preferred size, the available budget.

A more difficult task is however to provide the *User* with the facility of describing his or her general preferences, like for example: main preference is the cheapest option, second preference is the presence of a garden, and third preference is additional space, or preferences that cannot be easily parameterized, like for example: "if the flat is on the 3^{rd} floor or above the presence of an elevator is necessary". Therefore we have simplified the scenario by restricting our attention to a "prototypical user" for whom we defined a rulebase characterized by a given fixed set of parameters. The rulebase captures the "intelligence" of the *Broker* in understanding the preferences of a typical *User*, while the specific values of the parameters are input by the *User* at run-time using the form provided by the *Broker*. Eventually, an "experienced user" can still be provided with the possibility to visualize and update the rulebase. Moreover, if the user does not provide values for some of the preference parameters, then the corresponding rules will not be added to the generated rulebase.

In the second step the *Broker* will consult its internal database to retrieve a set of apartments which are relevant for the *User* request. In the third step, the *User* preferences (captured as the rulebase accompanied by a set of parameter-value pairs), together with the set of relevant apartments are submitted by the *Broker* to the reasoning service. Finally, the reasoning service returns to the *Broker* a set of available options. The *Broker* converts them into a meaningful presentation (eg. HTML) and sends it to the *User* for visualization in the browser.

4.2. Design

For the system design we have produced an UML activity diagram that shows the activities of the *User*, *Broker* and the reasoning service, shown in Fig. 6, as well as an UML sequence diagram detailing the interactions between them, shown in Fig.7. The reasoning service is named *DR Web service* and it is based on the *DR Reasoner* which is part of EMERALD reasoning platform. The system functioning can be better understood if we follow both diagrams.

The *User* connects to the Web site of the *Broker* by submitting the URL. The *Broker* delivers back to the *User* a Web page containing a form for preference input as a set of parameter-value pairs describing his options concerning the rental of an apartment. The *User* fills-in this form and submits it to the *Broker* Web site. Behind this form the *Broker* preprocesses the preferences and produces a rulebase represented in RuleML notation, following the example from [3]. The rulebase contains a set of defeasible rules that are used by the *Broker* to capture the generic preferences of a typical *User*. Differently from [3], the rulebase is kept separate from the set of parameter-value pairs (called *preferences* in Fig.7, interaction 3). This approach simplifies the operation of instantiating a particular scenario with the specific values input by the *User*. Intuitively, the rulebase represents the "intelligence" of the *Broker* in understanding and formalizing the requirements and preference of a prototypical user regarding the rental of apartments.

The *Broker* receives the *preferences*, converts them to RDF and publishes them to a temporary RDF file inside the Web server of the *Broker* Web site at a given URL. Next, the *Broker* determines an initial set of relevant apartments for the *User* request, maps this list to RDF using the method already shown in [3], and publishes it to a temporary RDF

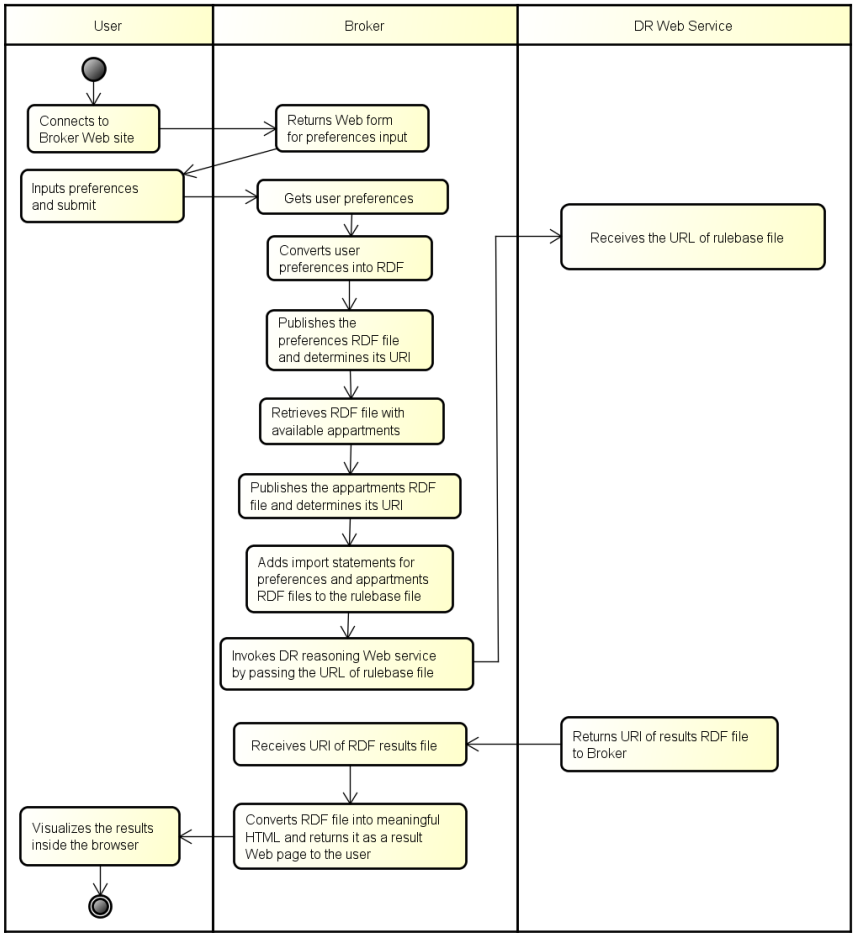


Fig. 6. Roles and activities in the brokering scenario.

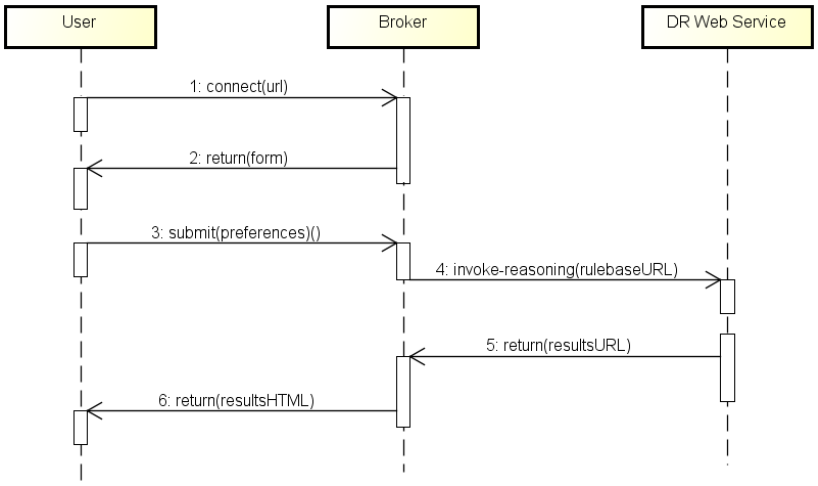


Fig. 7. Roles interaction in the brokering scenario.

file inside the Web server of the *Broker* Web site at a given URL. These two RDF files are included into the rulebase using the *rdf_import* attribute of RuleML, thus creating an updated rulebase that contains the generic rules as well as the two RDF files that specify the initial set of facts for the reasoning process.

Then the *Broker* invokes the *DR Web service* (interaction 4 in Fig.7) by passing it the *rulebase*. The *DR Web service* performs the reasoning task by using the EMERALD reasoning infrastructure, determines the output RDF file and returns its URL (*resultsURL*) to the *Broker* (interaction 5 in Fig.7). The *Broker* receives the URL, downloads the RDF file, converts it to meaningful HTML and returns the HTML (*resultsHTML*) to the *User* for visualization in the browser (interaction 6 in Fig.7).

4.3. Implementation

Following the above described design principles, a user-friendly Web site that hosts the *Broker* functionality was developed. Here, it is called *Broker Web site* since it is currently focused on this specific domain, however it can host any kind and any number of domains. The key issue behind this possibility is the fact that the site is hosted in an independent server. Having two separated servers, one for hosting the services such as the *DR Web Service* and one for hosting the registered parties such as the *Broker*, enables flexibility in the setting. In other words, it is important to preserve intact and unaffected the core framework which provides the services and at the same time provide an easy and efficient way for agent parties to communicate with human users. Adding new domains is in principle easily achievable, since it requires parsing the ontology of the input data and then using this ontology to author the appropriate rulebase for the end-user preferences. There exist graphical tools to achieve these, such as [14]] and [15]; however, in this paper we do not cover this aspect.

Fig. 8. Choosing domain in the GUI.

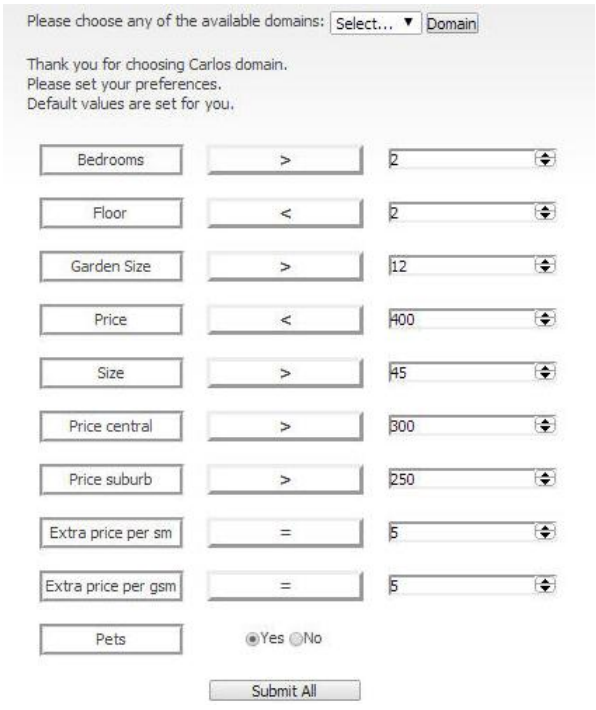
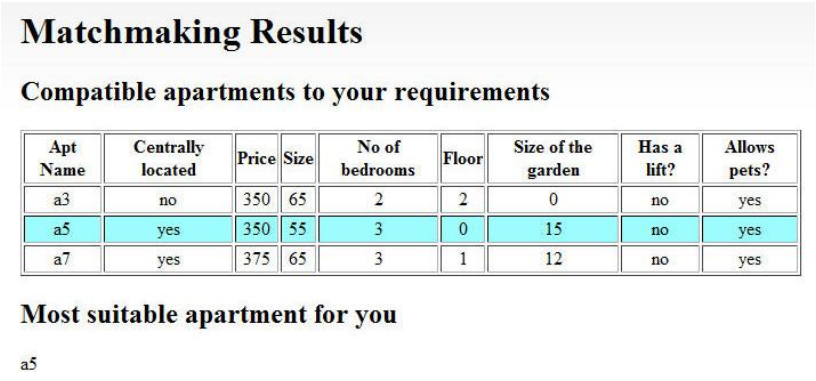


Fig. 9. Setting preferences.

In this study we used a new separated server for the newly developed GUI. This interface is implemented mainly in PHP, a server side language that allows all calculations and functions to be performed on the server leaving for the end user only what it is designed for him/her. Hence, when a user visits the Web site he/she has to choose from a dropdown menu the domain in which he/she is interested in; here it is Broker domain called Carlo in the implementation (Fig.8).

After the domain is chosen, the GUI is dynamically updated providing a list with all the available attributes for this domain (Fig.9). Hence, the user will be able to indicate his/her personal preferences, e.g. the number of bedrooms. For user convenience we provide some default values in the interface. When the user finishes, "Submit all" button is clicked, the data are collected and stored in the RDF data file that will be used in the *DR Web Service*. This file is stored in the server and each time users provide their preferred values appropriate PHP functions update the stored attribute values. More specifically, we have stored an RDF template data file, hence each time a new attribute values is set, the template is parsed and the value is stored under the appropriate instance.



Matchmaking Results

Compatible apartments to your requirements

Apt Name	Centrally located	Price	Size	No of bedrooms	Floor	Size of the garden	Has a lift?	Allows pets?
a3	no	350	65	2	2	0	no	yes
a5	yes	350	55	3	0	15	no	yes
a7	yes	375	65	3	1	12	no	yes

Most suitable apartment for you

a5

Fig. 10. GUI output.

Next, the updated RDF file, filled with the user's requirements and preferences, is used by the Broker in its request to the DR Web Service. As soon as the *DR Reasoner* processes the request, it sends back a reply containing the results in RDF syntax, here the compatible apartments to user's requirements, as well as the most suitable for him/her. The Broker on its side parses the RDF results and transforms then into HTML, in order to represent them in a user-friendly and easily understandable output to the user (Fig.10).

5. Proof-of-concept Scenario

In this section we present a proof-of-concept scenario of using the *Defeasible_Reasoning_Service* Web service for implementing the intelligent broker introduced in section 4. This scenario was adapted following [3], while it actually originates from [2].

We assume that a generic user with name Carlo has the following requirements for an apartment rental:

a) Apartment requirements
   - Minimum size: $45m^2$.
   - Minimum number of bedrooms: 2.
   - Maximum floor: 2.
   - Minimum garden size: 12.
   - Pets allowed: yes.

b) Price Requirements
   - Maximum budget: 400.
   - Maximum price for a central apartment: 300.
   - Maximum price for a suburb apartment: 250.
   - Extra price per additional apartment m^2: 5.
   - Extra price per additional garden m^2: 2.

c) Preferences
   - The cheapest apartment is mostly preferred.
   - The second option is the garden availability.
   - The third option is availability of additional space.

Now, differently from [3], in order to fit this scenario into our prototype system, we extracted from this set of requirements the generic rulebase representing the requirements of a typical user. Moreover, we captured the specific values of the requirements' parameters for Carlo as a separate RDF file shown in Fig. 12.

The rulebase that captures the Broker's intelligence as a set of defeasible rules is expressed in RuleML[3]. A snapshot of the rulebase is exemplified in Fig .11. The set of apartments available for rent that are registered with the the *Broker* is captured as an RDF file[4]. An example is shown in , shown in Fig. 12. This file, as well as the RDF file with parameter values from Fig. 12 are included into the rulebase using the *rdf_import* attribute of the *RuleML* root element of the rulebase file expressed in RuleML [3].

The *Broker* invokes the reasoning task by sending a request message to the *Defeasible_Reasoning_Service* Web service. This message is packaged using SOAP as shown in Fig. 13. The request specifies the URL of the rulebase file that is located on the *Broker*'s Web server, using the *inputRulemlPath* element.

The *Defeasible_Reasoning_Service* Web service replies with a response packaged as a SOAP file that contains a link to the RDF file with the results, published on the *Defeasible_Reasoning_Service* Web server. A part of the results file in this case is shown in Fig. 14. It contains the list of "acceptable" apartments – *carlo:acceptable* element with the *truthStatus* set to *defeasibly_proven_positive*, as well as the apartment which is chosen by Carlo – *carlo:rent* element. You can easily note that among the acceptable apartments, Carlo chose to rent apartment *a5*.

[3] The file is available at `http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo-rbase-flex-0.91.ruleml`.

[4] The file is available at `http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo-flex_ex.rdf`.

```
<RuleML
  rdf_export="export.rdf"
  rdf_export_classes="acceptable rent"
  rdf_import="http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo_ex.rdf"
  namespace declarations>
  ...
   <Assert>
      <Implies ruletype="defeasiblerule">
         <oid><Ind uri="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 uri="carlo:name"/>
                  <Var>x</Var>
               </slot></Atom></body>
      </Implies>
      <Implies ruletype="defeasiblerule">
         <oid><Ind uri="r2">r2</Ind></oid>
         <head>
            <Neg><Atom><op><Rel>acceptable</Rel></op>
                  <slot><Ind>apartment</Ind>
                     <Var>x</Var>
                  </slot></Atom></Neg></head>
         <body><And>
               <Atom><op><Rel uri="carlo:apartment"/></op>
                  <slot><Ind uri="carlo:name"/>
                     <Var>x</Var></slot>
                  <slot><Ind uri="carlo:bedrooms"/>
                     <Var>y</Var></slot></Atom>
               <Atom><op><Rel uri="carlo:requirement"/></op>
                  <slot><Ind uri="carlo:min-bedrooms"/>
                     <Var>mb</Var></slot></Atom>
               <Test><Expr>
                     <Fun in="yes">&lt;</Fun>
                     <Var>y</Var>
                     <Var>mb</Var></Expr></Test></And></body>
         <superior><Ind uri="r1"/></superior>
      </Implies>
      ...
      <Implies ruletype="defeasiblerule">
         <oid><Ind uri="r11">r11</Ind></oid>
         <head>
            <Atom><op><Rel>rent</Rel></op>
               <slot><Ind>apartment</Ind>
                  <Var>x</Var></slot></Atom></head>
         <body> <And>
               <Atom><op><Rel>cheapest</Rel></op>
                  <slot><Ind>apartment</Ind>
                     <Var>x</Var></slot></Atom>
               <Atom><op><Rel>largestGarden</Rel></op>
                  <slot><Ind>apartment</Ind>
                     <Var>x</Var></slot></Atom></And></body>
         <superior><Ind uri="r10"/></superior>
      </Implies>
      ...
   </Assert>
</RuleML>
```

Fig. 11. Rule base fragment.

```
<rdf:RDF ... declarations of namespaces>
...
<carlo:apartment rdf:about="&carlo_ex;a5">
   <carlo:bedrooms
     rdf:datatype="&xsd;integer">3</carlo:bedrooms>
   <carlo:central>yes</carlo:central>
   <carlo:floor
     rdf:datatype="&xsd;integer">0</carlo:floor>
   <carlo:gardenSize
     rdf:datatype="&xsd;integer">15</carlo:gardenSize>
   <carlo:lift>no</carlo:lift>
   <carlo:name>a5</carlo:name>
   <carlo:pets>yes</carlo:pets>
   <carlo:price
     rdf:datatype="&xsd;integer">350</carlo:price>
   <carlo:size
     rdf:datatype="&xsd;integer">55</carlo:size>
   <rdfs:label>a5</rdfs:label>
</carlo:apartment>
...
<carlo:requirement rdf:about="&carlo_ex;req1">
   <carlo:min-bedrooms
     rdf:datatype="&xsd;integer">2</carlo:min-bedrooms>
   <carlo:max-floor
     rdf:datatype="&xsd;integer">2</carlo:max-floor>
   <carlo:min-gardenSize
     rdf:datatype="&xsd;integer">12</carlo:min-gardenSize>
   <carlo:pets-req>yes</carlo:pets-req>
   <carlo:max-price
     rdf:datatype="&xsd;integer">400</carlo:max-price>
   <carlo:min-size
     rdf:datatype="&xsd;integer">45</carlo:min-size>
   <carlo:min-price-central
     rdf:datatype="&xsd;integer">300</carlo:min-price-central>
   <carlo:min-price-suburb
     rdf:datatype="&xsd;integer">250</carlo:min-price-suburb>
   <carlo:extra-price-per-sm
     rdf:datatype="&xsd;integer">5</carlo:extra-price-per-sm>
   <carlo:extra-price-per-gsm
     rdf:datatype="&xsd;integer">5</carlo:extra-price-per-gsm>
</carlo:requirement>
</rdf:RDF>
```

Fig. 12. RDF document for available apartments and parameter values.

```
<soapenv:Envelope ... declarations of namespaces>
  <soapenv:Header/>
  <soapenv:Body>
    <urn:DRReasoning
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <inputRulemlPath xsi:type="xsd:string">
       http://lpis.csd.auth.gr/systems/dr-device/carlo/carlo-rbase-flex-0.91.ruleml
      </inputRulemlPath>
    </urn:DRReasoning>
  </soapenv:Body>
</soapenv:Envelope>
```

Fig. 13. SOAP message used by the *Broker* to request a reasoning task to the *DR Web service*.

```
<rdf:RDF ... declarations of namespaces>
   ...
   <export:acceptable rdf:about="&export;acceptable4">
      <export:apartment>a4</export:apartment>
      <defeasible:truthStatus>defeasibly_proven_negative</defeasible:truthStatus>
   </export:acceptable>
   <export:acceptable rdf:about="&export;acceptable2">
      <export:apartment>a5</export:apartment>
      <defeasible:truthStatus>defeasibly_proven_positive</defeasible:truthStatus>
   </export:acceptable>
   ...
   <export:rent rdf:about="&export;rent1">
      <export:apartment>a5</export:apartment>
      <defeasible:truthStatus>defeasibly_proven_positive</defeasible:truthStatus>
   </export:rent>
</rdf:RDF>
```

Fig. 14. RDF file with the reasoning results.

6. Conclusions

In this paper we proposed an approach for reusing agent-based reasoning capabilities by making them available for invocation as Web services. Our approach is supported by a prototype system that provides an online brokering function in the domain of apartments rental. Firstly, we proposed an extension of the EMERALD framework for agent based reasoning services with a Web service interface. Then we considered an intelligent brokering proof-of-concept scenario that involves the defeasible reasoner available in the EMERALD framework.

Our approach is general enough so it can be applied to other types of reasoners, as well as to other problem domains. As future work we plan to (i) extend the system by providing other types of reasoning services over the Web such as the deductive reasoner of EMERALD [17]; (ii) experiment with more complex scenarios that require multiple and possibly different reasoning tasks; and (iii) extend the Web services to serve multiple operations regarding the reasoning tasks, such as proof explanation ([19]); (iv) enhance the description of reasoning Web services with metadata (including for example attributes like trust, performance, quality, reasoning type, and representation formalism) for allowing clients to search and select the most appropriate reasoner for their task in hand.

Finally, a longer term goal is to provide an open, flexible and customizable Web-agent platform for hosting various agent communities and activities over the Web, such as negotiation and auction platforms for e-business applications. They can benefit from the reasoning services provided by our Web-agent-based extension of EMERALD, for example by declarative representation of private agent strategies, as well as of public negotiation mechanisms for agent-based negotiation activities [28,9,21].

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures and Applications. Springer (2004)
2. Antoniou, G., van Harmelen, F.: A Semantic Web Primer – Second Edition. MIT Press (2008)
3. Bassiliades, N., Antoniou, G., Vlahavas, I.P.: A defeasible logic reasoner for the semantic web. International Journal on Semantic Web and Information Systems 3(1), 1–41 (2006)

4. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. John Wiley & Sons Ltd (2007)
5. Boley, H., Paschke, A.: Rule responder agents framework and instantiations. In: Elçi, A., Koné, M.T., Orgun, M.A. (eds.) Semantic Agent Systems, Studies in Computational Intelligence, vol. 344, pp. 3–23. Springer (2011)
6. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex PO19 8SQ, England (2007)
7. Bădică, C., Budimac, Z., Burkhard, H.D., Ivanović, M.: Software agents: Languages, tools, platforms. Computer Science and Information Systems 8(2), 255–298 (2011)
8. Dastani, M., Birna Riemsdijk, M., Meyer, J.J.C.: Programming multi-agent systems in 3apl. In: Bordini, R., Dastani, M., Dix, J., Fallah Seghrouchni, A. (eds.) Multi-agent programming: Languages platforms and applications, Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 15, pp. 39–67. Springer, US (2005)
9. Dobriceanu, A., Biscu, L., Bădică, A., Bădică, C.: The design and implementation of an agent-based auction service. International Journal of Agent-Oriented Software Engineering 3(2/3), 116–134 (2009)
10. FIPA: The foundation for intelligent physical agents (fipa): Fipa communicative act library specification (2002), retrieved November 29, 2012 from `http://www.fipa.org/specs/fipa00037/`
11. Gelder, A.V., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM 38(3), 620–650 (1991)
12. Giarratano, J.C., Riley, G.D.: Expert Systems: Principles and Programming, Fourth Edition. Course Technology (2004)
13. JADE: Jade web services integration gateway (wsig) guide (2012), retrieved March 23, 2013 from `http://jade.tilab.com/doc/tutorials/WSIG_Guide.pdf`
14. Kontopoulos, E., Bassiliades, N., Antoniou, G., Seridou, A.: Visual modeling of defeasible logic rules with dr-vismo. International Journal of Artificial Intelligence Tools (IJAIT) 17(5), 903–924 (2008)
15. Kontopoulos, E., Zetta, T., Bassiliades, N.: Semantically-enhanced authoring of defeasible logic rule bases in the semantic web. In: Proc. 2nd International Conference on Web Intelligence, Mining and Semantics (WIMS'12). Craiova, Romania, June 13-15, 2012. pp. 489–492, Article 56. ACM (2012)
16. Kravari, K., Kontopoulos, E., Bassiliades, N.: A trusted defeasible reasoning service for brokering agents in the semantic web. In: Papadopoulos, G.A., Bădică, C. (eds.) Intelligent Distributed Computing III, Proc. 3rd International Symposium on Intelligent Distributed Computing, IDC 2009. Studies in Computational Intelligence, vol. 237, pp. 243–248. Springer (2009)
17. Kravari, K., Kontopoulos, E., Bassiliades, N.: Emerald: A multi-agent system for knowledge-based reasoning interoperability in the semantic web. In: Konstantopoulos, S., Perantonis, S.J., Karkaletsis, V., Spyropoulos, C.D., Vouros, G.A. (eds.) Artificial Intelligence: Theories, Models and Applications, Proc. 6th Hellenic Conference on AI, SETN 2010. Lecture Notes in Computer Science, vol. 6040, pp. 173–182. Springer (2010)
18. Kravari, K., Kontopoulos, E., Bassiliades, N.: Trusted reasoning services for semantic web agents. Informatica (Slovenia) 34(4), 429–440 (2010)
19. Kravari, K., Papatheodorou, K., Antoniou, G., Bassiliades, N.: Extending a multi-agent reasoning interoperability framework with services for the semantic web logic and proof layers. In: Bassiliades, N., Governatori, G., Paschke, A. (eds.) Rule-Based Reasoning, Programming, and Applications, Lecture Notes in Computer Science, vol. 6826, pp. 29–43. Springer Berlin Heidelberg (2011)
20. Kwon, O., Im, G., Lee, K.: An agent-based web service approach for supply chain collaboration. Scientia Iranica 18(6), 1545–1552 (2011)

21. Muscar, A., Bădică, C.: Towards a declarative framework for the specification of agent-driven auctions. Engineering Intelligent Systems 21(2-3), 642–651 (2013)
22. nerdErg: The one ring project, nerderg pty ltd 2012 (2013), retrieved June 10, 2013 from http://nerderg.com/One+Ring
23. Nute, D.: Defeasible reasoning. In: Proc. 20th International Conference on Systems Science. pp. 470–477. IEEE Press (1987)
24. Purvis, M., Cranefield, S., Nowostawski, M., Carter, D.: Opal: A multi-level infrastructure for agent-oriented software development. In: Information Science Discussion Paper Series No. 2002/01. University of Otago, New Zealand (2002)
25. Ryzko, D., Radziszewska, W.: Integration between web services and multi-agent systems with applications for multi-commodity markets. In: Kaleta, M., Traczyk, T. (eds.) Modeling Multi-commodity Trade: Information Exchange Methods, Advances in Intelligent and Soft Computing, vol. 121, pp. 65–77. Springer Berlin Heidelberg (2012)
26. Sabater, J., Sierra, C.: Review on computational trust and reputation models. Artificial Intelligence Review 24(1), 33–60 (2005)
27. Skillen, K.L., Chen, L., Nugent, C.D., Donnelly, M.P., Burns, W., Solheim, I.: Ontological user modelling and semantic rule-based reasoning for personalisation of help-on-demand services in pervasive environments. Future Generation Computer Systems 34, 97–109 (2014)
28. Skylogiannis, T., Antoniou, G., Bassiliades, N., Governatori, G., Bikakis, A.: Dr-negotiate - a system for automated agent negotiation with defeasible logic-based strategies. Data & Knowledge Engineering 63(2), 362–380 (2007)
29. Su, C.J., Peng, C.W.: Multi-agent ontology-based web 2.0 platform for medical rehabilitation. Expert Systems with Applications 39(12), 10311–10323 (2012)
30. Walton, C.D.: Agency and the Semantic Web. Oxford University Press (2007)
31. Wang, G., Wong, T., Wang, X.: An ontology based approach to organize multi-agent assisted supply chain negotiations. Computers & Industrial Engineering 65(1), 2–15 (2013)
32. Wang, M., Purvis, M., Nowostawski, M.: 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), Washington, DC. pp. 58–64 (2005)
33. Wooldridge, M.: An Introduction to MultiAgent Systems – Second Edition. John Wiley & Sons (2009)