Agents and Knowledge Interoperability in the Semantic Web Era

Nick Bassiliades
Dept. of Informatics
Aristotle University of Thessaloniki
Greece
+302310997913
nbassili@csd.auth.gr

ABSTRACT
This tutorial will discuss about issues, technologies and tools that concern the way that the Semantic Web affects knowledge and information interchange among intelligent agents in multi-agent systems, as well as reasoning interoperability. First, the tutorial will discuss how semantic web rules and ontologies interact with each other in order to be used as the agent’s internal knowledge base for environment awareness and decision making. Then, interoperability between reasoning systems for agents will be discussed. The issues involved in all the previous discussion will be exemplified using actual implemented tools for semantic web reasoning in multi-agents systems.

Categories and Subject Descriptors
I.2.11 [Artificial Intelligence]: Distributed Artificial Intelligence – intelligent agents, languages and structures, multiagent systems.
I.2.3 [Artificial Intelligence]: Deduction and Theorem Proving - inference engines, nonmonotonic reasoning.
D.2.12 [Software Engineering]: Interoperability.

General Terms
Algorithms, Design, Standardization, Languages, Theory.

Keywords
Multiagent systems, Semantic Web, Rules, Ontologies, Reasoning Engines, Knowledge Interoperability.

1. INTRODUCTION
The Semantic Web (SW) [8] is a rapidly evolving extension of the WWW that derives from Sir Tim Berners-Lee’s vision of a universal medium for data, information and knowledge exchange. The SW provides the possibility for both people and machines to understand, and act upon, Web content with high precision. The current evolution of its technologies, namely metadata in the form of RDF [46] and ontologies in the form of the OWL 2 language [53], offer interoperability at the data and the knowledge level. Research efforts are now shifting towards the higher layers of logic and proofs.

The development of the two latter layers is critical, since they will allow systems to infer new knowledge from existing information, assisting them in explaining their actions, sources and beliefs and, eventually, increasing user trust towards the Semantic Web. Researchers are focusing mainly on the integration of rules and ontologies via the OWL2 RL approach [47], which comprises the intersection of Horn logic and description logics, or through efforts for standardizing rule representation in the Semantic Web, e.g. via the Semantic Web Rule Language (SWRL) [24], the RuleML markup initiative [9] and the Rule Interchange Format (RIF) [10].

The languages of RDF and OWL dialects/profiles can be viewed as specializations of predicate logic. One justification for the existence of such specialized languages is that they provide a syntax that fits well with the intended use for the Web. The other major justification is that they define reasonable subsets of logic, since there is a trade-off between the expressive power and the computational complexity of certain logics: the more expressive the language, the less efficient the corresponding proof systems. OWL variants correspond to a description logic, a subset of predicate logic for which efficient proof systems exist.

Another subset of predicate logic with efficient proof systems comprises the rule systems, also known as Horn logic or definite logic programs. Description logics and Horn logic are orthogonal; neither of them is a subset of the other [17]. Actually, they are both needed in expressing different kinds of knowledge in the Semantic Web [1], thus the need for two different layers in the SW architecture. Furthermore, different reasoning engines (or reasoners) are used for the above two subsets of logic.

So, the first part of this tutorial will discuss how semantic web rules and ontologies interact with each other in order to be used as the agent’s internal knowledge base for environment awareness and decision making. Various ways of interaction, interoperation and integration will be discussed.

Intelligent agents (IA – an autonomous software program that carries out tasks on behalf of users, exhibiting some form of intelligence) are considered the most prominent means towards realizing the SW vision [19]. 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 formalism; even in the light of the standardization efforts of RIF and RuleML, these initiatives are developing a family of rule dialects, which share a common syntax and some very basic semantics for their core constructs, but whose full semantics can differ a lot. 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 interoperability.

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 is usually able to implement the semantics a single logic/rule dialect and, consequently, agents would require a common interchange language. Since generating a translation schema from one (rule) dialect into the other (e.g. RIF – Rule Interchange Format [10]) 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, where reasoning services are wrapped in IAs, has been proposed in [32] and is presented in the second part of this tutorial. 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 this line of research, thus a variety of popular inference services that conform to various types of logics has been implemented. Some of the incorporated reasoning engines are also reviewed in this tutorial.

2. RULES AND ONTOLOGIES
The development of Semantic Web proceeds in layers where each layer is built on top of the others. Currently, the ontology layer has reached a sufficient level of maturity, having OWL as the basic form for ontology definition. The next step is to move on the higher levels of logic and proof, which are built on top of the ontology layer, where rules now are considered as the primary key, since (a) they can serve as extensions of, or alternatives to, DL based ontology languages and (b) they can be used to develop declarative systems on top of ontologies.

Although there is a lot of debate about the suitability of Logic Programming (LP) in the domain of the Semantic Web, many research efforts have been focused on the mapping of DLs into LP or on the combination of DLs and LP in order to overcome the shortcomings that emerged during the development of practical OWL applications ([44], [42]). Such a mapping or combination is important for many aspects of the Semantic Web, such as:

- **Querying**: It is interesting to consider combining DLs with the rule paradigm in order to state expressive instance queries, since DL reasoning engines have rather a low ABOX reasoning and querying performance ([23], [21]).
- **Non-monotonicity**: DLs follow the principle of the open world assumption. However, sometimes it is preferable to introduce non-monotonicity in the DLs ([42], e.g. the notion of negation as failure in logic programs.
- **DLs’ expressivity**: Rules can serve as extensions of description logic based ontology languages ([22]).
- **Integrity constraints**: Sometimes is useful to be able to define integrity constraints, i.e. constraints over the ABOX.

Concerning the mapping of DL to LP, Grosof, et al. ([17]) define the intersection of LP and DL, namely the Description Logic Programs (DLP). Actually, DLP is the most expressive sublanguage of OWL DL that can be efficiently mapped to Datalog and it is simpler than OWL Lite. In that way, it is possible to interoperate between rules and ontologies, transforming LP to DL and vice versa.

The OWL2 RL profile of OWL2 [47] is based on DLP and enables interaction between description logics and rules: it is the largest syntactic fragment of OWL2 DL that is implementable using rules. This is a very important feature, as rules can efficiently be run in parallel, allowing for highly scalable reasoning implementations. Many of the most scalable reasoners for Semantic Web languages implement OWL2 RL or a very similar language called pD* or OWL-Lhorst (see section 2.1.1). The set of rules that have to be implemented is published as part of the OWL2 RL specification.

While DLP and OWL2 RL is the intersection of LP and DL, the OWL Flight [11] is an ontology language based totally on the LP subset of OWL. It is inspired by DLP and imposes certain extensions in the area of datatypes, database-style constraints, such as cardinality and value constraints, and meta-modeling. OWL Flight restricts the OWL syntax such that it falls in the Datalog fragment and thus query answering can be done using an LP implementation.

The major flaw of the mapping approaches is the fact that there is not an unrestricted mapping of OWL semantics into the rule paradigm, and thus the resulting languages have restricted semantics, handling a subset of OWL DL. To solve this expressivity problem, many research efforts have been focused on the combination of DL and LP. Such a combination is realized following either a homogeneous or a hybrid approach [2].

2.1 Homogeneous approach
The homogeneous approaches treat rule and ontology predicates homogeneously, as a new single logic language. The general idea is that the rules can use unary and binary predicates from the ontology (i.e., classes and properties) as well as predicates that occur only in rules (rules predicates). In order to maintain the decidability of the integrated language, there is usually a safety condition that restricts variables occurring in the head of a rule to those that occur in at least one positive rule predicate in the body of the rule. Intuitively, in homogeneous approaches, the OWL semantics are mapped into a rule-based formalism, e.g. Datalog rules that coexist in the KB with rule predicates, enhancing the expressivity. The homogeneous approaches can be used either for building rule programs on top of ontologies or ontologies on top of rules. Thus, a new reasoner is needed, able to handle the new homogeneous language that emerges ([50], [20], [41]). In fact, the mapping
approaches we described previously ([26], [42]) can be consid-
ered as the first step for building a homogeneous system.

Another proposal is the Semantic Web Rule Language (SWRL) [24], a non-safe approach to the integration of rules and DLs in which rules are interpreted under the classical first order logic semantics. The addition of this kind of rules to DLs leads to undecidability of reasoning.

In the following section we describe the basic principles of the entailment-based OWL reasoning (EBOR) paradigm that enables the materialization of OWL semantics into the KB of a rule engine using OWL entailment (inference) rules. The EBOR paradigm can be considered as the first step in realizing a homogeneous combination of OWL and rules in order to build rule programs on top of ontologies, as well as ontologies on top of rule programs, since the rule program coexists with the inference rules in the rule base, and thus, the rule execution is interleaved with the inference procedure.

2.1.1 Entailment-Based OWL Reasoning

The EBOR paradigm is realized by using a rule engine as the inference engine in the architecture of Figure 1, implementing RDF/OWL entailment rules ([18], [26]).

![Figure 1. The abstract architecture of an OWL reasoner.](image)

The semantics of RDF and RDFS can be captured using entailments rules [18], which are rules that denote the information that should be derived based on existing one. Intuitively, an entailment rule is an if-then rule that denotes the knowledge that should be inferred (rule head) based on existing knowledge (rule body). The body consists of RDF statements, where variables can occupy any of the three possible positions in the triple (that of a subject, of a predicate, or of an object). The head of the rule comprises of one or more consequences, each of which represents in turn an RDF statement. The consequences may not contain free variables, i.e. such that are not used within the body of the rule. The list of the RDF/RDFS entailments is defined in [18]. We give as an example the rdfs9 entailment rule using the N-Triple notation.

\[
\text{if } <s> <p> <x> \land
<x> <rdfs:type> <owl:TransitiveProperty> .
\text{then} <x> <p> <z>. \]

The rdfs9 entailment rule actually defines the subsumption characteristic of the rdfs:subClassOf property: if there is an instance \(<x>\) defined to belong to the class \(<c>\), and \(<c>\) is defined as a subclass of the class \(<d>\), then \(<x>\) is also of type \(<d>\).

Although there is a complete set of RDF and RDFS entailments [18], such a complete set does not exist for OWL due to the great degree of expressiveness. Horst ([25]) defines the pD* semantics as a weakened variant of OWL Full and then in [26] the pD* semantics were extended to apply to a larger subset of the OWL vocabulary. More precisely, the pD* semantics can be realized by 23 entailment rules and 2 inconsistency rules. To exemplify, we present the rdfs9 entailment that handles the values of transitive properties that are defined using the owl:TransitiveProperty OWL construct [1].

\[
\text{if } <p> <rdfs.type> <owl:TransitiveProperty> .
<s> <p> <x>. \land
<x> <p> <z>. \land
\text{then} <s> <p> <z>.}
\]

The pD* semantics extend RDFs and they are defined in a way analogous to the if-semantics of RDFS, leading to simple entailment rules that can be used to extend RDF reasoners. In other words, we do not obtain the full power of OWL’s iff semantics in pD* semantics. Instead, they represent a reasonable interpretation that is useful for drawing conclusions about instances in an ontology and that lead to simple entailment rules with a relatively low computational complexity (consistency is in P and entailment is NP-complete, and in P if there are not blank nodes in the target graph).

In the EBOR paradigm, the asserted knowledge, that is the knowledge that stems directly from the ontology definition, is mapped into an internal rule engine representation format, and inference rules, which are expressed in the language of the rule engine, are applied in order to deduce new knowledge or to check the consistency of the ontology, based on OWL entailments. To exemplify, let S be the set of triples of an ontology, where \(S = \{<A \text{ subClassOf } B>, <x \text{ type } A>\}\). By implementing the rdfs9 entailment rule, we get that \(S = \{<A \text{ subClassOf } B>, <x \text{ type } A>, <x \text{ type } B>\}\).

Therefore, for the development of an EBOR system, three issues should be tackled:

- **Ontology mapping**: An EBOR system should define a mapping procedure of the ontological knowledge into the KB of the rule engine that uses. Usually, such a mapping procedure is performed over the ontology triples. The purpose of this phase is to generate an internal, rule engine-specific representation of the ontological information where the entailment rules will be applied on.
- **Inferencing process**: An EBOR system should implement the desirable number of entailment rules expressed in the engine’s rule language. This phase actually defines the reasoning completeness of the EBOR system that usually comes with implementations of different expressiveness according to the number of entailments that are implemented.
- **Query support**: An EBOR system should be able to answer queries about the semantic derivations of its KB. Since the core system is a rule engine, the query infrastructure is implemented with query rules. These query rules follow either the rule language of the underlying rule engine, or they have a standard-based syntax ([55], [45]).

There are two approaches for the development of an EBOR system, namely the extended and the native approach. An extended entailment-based OWL reasoning (E-EBOR) system is built on top of an existing, general purpose rule engine that augments it with the ability of manipulating ontological information. This incorporates the ability of transforming the ontological information into facts and populating its rule base with the appropriate inference rules. A native entailment-based OWL reasoning (N-
EBOR) system is built from scratch and draws conclusions directly on the OWL data model. Each approach has advantages and disadvantages and the right choice depends on the requirements of the application. More specifically:

- **Reasoning Performance**: Since an N-EBOR system is built directly upon the OWL data model, it has increased reasoning performance, as far as speed issues are concerned, comparing it to the E-EBOR paradigm that does not apply any optimization in the way it handles the ontological information. Thus, an N-EBOR system is an appropriate choice in the cases where reasoning speed is a critical requirement.

- **Ontology Utilization**: The use of an E-EBOR system gives the opportunity to efficiently utilize the ontology information by building rule-based applications. Ontologies can be inserted into the system and, after the application of the inference rules, user-defined rules can operate over the inferred knowledge. In that way, an EBOR system built on top of a general purpose rule engine gives the opportunity to reuse the practicality, efficiency and optimized techniques that rule engines have obtained throughout the years of their development. On the other hand, an N-EBOR system built from scratch tends to throw away decades of research and development on efficient and robust rule engines. Therefore, an E-EBOR system has increased capabilities concerning post-reasoning utilization of ontological information into rule programs.

In the following subsection we describe in more detail one such EBOR system, O-DEVICE [40], that has been developed by a team led by the author.

2.1.2 The O-DEVICE System

O-DEVICE is built on top of the CLIPS production rule engine [48]. Its reasoning process is characterized by the transformation of ontological information into the object-oriented model of the COOL language of CLIPS and the application of inference production rules over the generated object-oriented schema.

CLIPS is a RETE-based production rule engine written in C that was developed in 1985 by NASA's Johnson Space Center and it has undergone continual refinement and improvement ever since. Today it is widely used throughout the government, industry and academia. One of the most interesting capabilities of CLIPS is that integrates the production rule paradigm with the object-oriented model, which can be defined using the COOL (CLIPS Object-Oriented Language) language of CLIPS. In that way, classes, attributes and objects can be matched on the production rule conditions (LHS), as well as to be altered on rules actions (RHS).

O-DEVICE transforms OWL ontologies into triples and applies a set of transformation rules in order to generate a COOL-based object-oriented schema of classes, attributes (slots) and objects. For example, the Region class and the subRegionOf property of the region ontology are transformed into a COOL defclass construct and the ontology instances into COOL objects, as follows:

```clips
(defclass Region
  (is-a owl:Thing)
  (subRegionOf (type INSTANCE-NAME)))

(make-instance region1 of Region)
(make-instance region2 of Region)
(make-instance region3 of Region)
```

The object-oriented model that is generated in O-DEVICE is fully compliant with object-oriented principles. To exemplify, consider the case where an ontology instance x is defined to belong to more than one classes, such as:

```clips
<x> <rdf:type> <Class1> .
<x> <rdf:type> <Class2> .
```

Since O-DEVICE is based on object-oriented principles, such a definition is not allowed, since every object can have only one class type. O-DEVICE handles this case by creating a subclass T of the two classes and defines x to be an instance of T, that is:

```clips
(defclass T  
  (is-a Class1 Class2))
(make-instance x of T)
```

The reasoning procedure of O-DEVICE separates the TBOX and the ABOX reasoning activities, using static production rules for the former, whereas it follows a template-based methodology for the latter, generating domain-dependent inference rules according to the degree of the expressiveness of the loaded ontology. To exemplify, the template rule for property transitivity (rdfp4) is defined in O-DEVICE as:

```clips
(defrule <rule-name>
  (object (is-a <pd>) (name ?o1) (<p> $? ?o2 $?))
  =>
  (bind $?v1 (send ?o1 get-<p>))
  (bind $?v2 (send ?o2 get-<p>))
  (send ?o1 put-<p> (union $?v1 $?v2)))
```

Analogous rules are used for the other properties that are not transitive. The reasoning procedure of O-DEVICE transforms a deductive rule into a production rule, with the use of a deductive rule language. Thus, the same transformation rules are transformed into a COOL defcl

```clips
(defrule subRegionOf
  (object (is-a Region) (name ?o1) 
  (subRegionOf <p> ?o2 ?))
  =>
  (bind $?v1 (send ?o1 get-subRegionOf))
  (bind $?v2 (send ?o2 get-subRegionOf))
  (send ?o1 put-subRegionOf (union $?v1 $?v2)))
```

The reasoning procedure of O-DEVICE separates the TBOX and the ABOX reasoning activities, using static production rules for the former, whereas it follows a template-based methodology for the latter, generating domain-dependent inference rules according to the degree of the expressiveness of the loaded ontology. To exemplify, the template rule for property transitivity (rdfp4) is defined in O-DEVICE as:

```clips
(defrule <rule-name>
  (object (is-a <pd>) (name ?o1) (<p> $? ?o2 $?))
  =>
  (bind $?v1 (send ?o1 get-<p>))
  (bind $?v2 (send ?o2 get-<p>))
  (send ?o1 put-<p> (union $?v1 $?v2)))
```

The deductive rule language of O-DEVICE supports querying over OWL instances represented as objects. The conclusions of deductive rules represent derived classes, whose objects are generated by evaluating these rules over the current set of objects. Each deductive rule is implemented as a CLIPS production rule that inserts a derived object when the condition of the deductive rule is satisfied. More specifically, the query for the instances of the Region class that have the region1 instance in their subRegionOf property is defined as:

```clips
(deductive rule region-instances
  (?id<- (Region (subRegionOf $? [region1] ?))))
  =>
  (DERIVED (result ?id)))
```

The query denotes that the objects that match the LHS of the rule will be collected as property values in the result slot of the objects of the DERIVED class. Thus, the query will generate two objects
of the DERIVED class with an arbitrary OID, with the region2 and region3 instances in their result slot.

2.2 Hybrid approach

The hybrid combination follows a modular architecture of two subsystems, each of which deals with a distinct portion of the knowledge base. More specifically, it combines the reasoning capabilities of a DL reasoner and the rule execution capabilities of a rule engine in order to define rules on top of the ontological information. Rule and ontology predicates are strictly separated and the ontology predicates can be used as constraints in rules. The hybrid approaches can be further classified into bidirectional and unidirectional, according to whether the derived knowledge flows from the rule module to the DL module or not. In the former case, DL constraints can be used in the head of the rules and thus, the ontological knowledge is altered, allowing the development of ontologies on top of rules ([13], [56], [27]). In the latter case, the information flows only from the DL component to the rule component by allowing only rule predicates to be used in rule heads and thus the ontological information remains unchanged ([34], [49], [12]).

In its simplest form, the unidirectional approach can be implemented as a one-time mapping of the results of the external OWL reasoner to the data model that the rule engine supports, e.g. on triple-based facts [28][29]. In that way, the rule engine can operate without calling further the ontology reasoner, since all the ontological knowledge exists in its KB. One such unidirectional one-time mapping approach has been implemented by a team led by the author in the CLIPS–OWL framework [39], which is described in the subsection 2.2.1.

Finally, in many practical applications, the terminological vocabulary (TBox) is not (or rarely) modified by applications at run-time, whereas the extensional part, namely ontology individuals (ABox), is continuously modified or enriched with new individuals. In such scenarios, there is a need for scalable implementations, whereas DL reasoners do not scale-up well (worst-case complexity is exponential). The DLE framework, presented in subsection 2.2.2, was developed by a team led by the author [37], to address the above efficiency problem by combining the completeness of DL reasoners for TBox reasoning with the low complexity of a production rule engine for ABox reasoning.

2.2.1 The CLIPS–OWL Framework

CLIPS–OWL [39] enables the CLIPS production rule engine [48] to represent the extensional results of DL reasoning on OWL ontologies in the form of Object-Oriented (OO) models. The purpose of this transformation is to allow CLIPS to use these OO models as static query models that are able to answer extensional queries directly by the RETE [14] reasoning engine. This enables the development of custom CLIPS production rule programs, without interfacing at runtime the external DL reasoner. In that way, any CLIPS-based application may enhance its functionality by incorporating ontological knowledge without modifying the architecture of the CLIPS rule engine. CLIPS–OWL has been implemented using the Pellet DL reasoner [51] and the CLIPS Object-Oriented Language (COOL).

The object-oriented data model that CLIPS–OWL follows is exactly the same as the one presented for O-DEVICE in the previous section. What is different is the reasoning process, which instead of being performed by a rule engine, thus being incomplete by nature, it is performed by a DL-reasoner, i.e. it is complete.

The main functionality of CLIPS–OWL is to allow decision-making rule-based applications in CLIPS to run on top of ontological knowledge. It could be argued that such rule programs can be also developed using existing systems, such as RACER or Pellet that provide the infrastructure to run a set of rules on top of ontological knowledge. This argument is true, but it should be noted that the intention behind CLIPS–OWL is not to substitute or outperform existing implementations. CLIPS–OWL is an engine-specific framework, allowing CLIPS to be used as a production rule engine in order to develop rule-based applications on top of OWL ontologies. This capability is considered quite important, since CLIPS is a highly efficient general-purpose production rule engine with many years of development, having been widely used throughout the government, industry and academia. Therefore, it is capable for building large and complex rule-based applications using traditional production rule engine features (conflict resolution strategies, message dispatching, salience, modules and many others).

The advantages of the one-time mapping architecture in CLIPS-OWL are the following:

- In a hybrid architecture, the rule engine is able to perform queries to the external reasoner only about the resources that exist in the KB, since they have to be matched by the rules. Therefore, the development of a rule program in a hybrid architecture requires the definition of a common vocabulary that would exist both in the rule engine and in the ontology. In mapping-oriented architectures, such as in CLIPS–OWL, the complete ontological knowledge is accessible and can be used by the rules, making more practical and straightforward the development of rule-based applications that use ontological knowledge.
- The implementation of a hybrid architecture using an existing general-purpose rule engine, such as CLIPS, is a very complex task compared to a rule engine that has been developed for this specific task. Any effort to embed the results of the external ontology reasoner calls directly into the RETE algorithm of CLIPS requires a great amount of changes to the initial infrastructure of CLIPS, resulting in a completely different architecture with unspecified side effects. Such an attempt might throw away decades of development on the efficient and robust RETE-based CLIPS rule engine.
- The development of a protocol for the notification of the rule engine about potential changes to the ontological knowledge will have to be too complex, especially in a RETE-based production rule engine, since the RETE algorithm maintains intermediate caches in order to implement an incremental pattern-matching activity.

2.2.2 The DLE Framework

In many practical applications, there is usually a shared terminological (TBox) vocabulary on which the ontology individuals (ABox) are defined. The TBox is not (or rarely) modified by applications at run-time, whereas the extensional part is continuously modified or enriched with new individual assertions.

In such scenarios, the ontology vocabulary is partitioned, that is, the vocabulary of classes and properties is separated from the individuals vocabulary, since the classes and properties are not used at the same time as individuals. Furthermore, the extensional knowledge usually involves large number of individual assertions and therefore, there is a need for scalable implementations. The DLE framework targets at the efficient execution of the individ-
ual-related RDF triple-based inference rules in such application domains by defining a framework for the combination of a DL reasoner and a production rule engine. The motivation of such a combination is summarized in the following.

- **Degree of TBox completeness.** Many practical ontology reasoners [4], [5], [8] and [6] implement forward-chaining inference rules in the form of RDF triple-based rules [13] for TBox and ABox reasoning, following the RDF Model Theory [11]. These approaches, like DLP, have limited TBox reasoning capabilities, since they are based on rules. For example, we have observed that they are unable to derive that two properties \( p \) and \( q \) are equivalent if \( p \) and \( q \) are both the inverse properties of \( g \), because they do not implement the corresponding entailment rule.5 The DLE framework, instead of using rules for TBox reasoning, it uses a DL reasoner for TBox reasoning completeness.

- **ABox reasoning performance.** The ABox reasoning procedure in DLE is based on instantiated individual-related inference rules. The experimental evaluation of DLE with a number of rule engines [9] has shown that the instantiated rules are executed faster with less memory requirements, compared to the execution of predefined, generic rules in the same rule engine.

In DLEJena [38], an implementation of the DLE framework for the OWL 2 RL profile, using the Pellet DL reasoner and the forward-chaining rule engine of Jena has been presented. DLEJena combines the reasoning paradigm that is based on the execution of RDF triple-based entailment rules in a forward-chaining rule engine [36] and the transformation paradigm of an ontology into a set of instantiated instance-related entailment rules. It should be noticed that the instantiation of these rules is based on TBox reasoning in order to allow DJEJena to handle an arbitrary number of entailments beyond the OWL 2 RL profile.

### 2.2.2.1 OWL 2 RL/RDF rule classification

DLEJena is based on the classification of the OWL 2 RL/RDF rules in three categories, according to the semantic conditions that are involved in each rule. This classification targets at the identification of the schema-related rules, whose semantics are implemented by Pellet, and the individual-related rules that are implemented as templates or ordinary Jena rules. The characteristics of each category are as follows:

- **Terminological rules.** The rules that deduce class and property relationships are referred to as terminological. These rules are not implemented in DLEJena and their semantics are handled by Pellet instead. For example, the rule that handles the subclass transitivity:

  \[
  (?c1 rdfs:subClassOf ?c2) \rightarrow (?c1 rdfs:subClassOf ?c3)
  \]

- **Hybrid Rules.** These rules deduce individual relationships by matching both TBox and ABox information in their body. As we explain later, these rules are defined through templates in DLEJena that generate instantiated ABox rules. For example, the rule that defines the inverse property relationship:

  \[
  (?p owl:inverseOf ?q) \rightarrow (?y ?q ?x)
  \]

- **Exceptional rules.** Finally, the rules that deduce ABox relationships by matching only ABox information in their body are referred to as exceptional. These rules are expressed in DLEJena directly as Jena rules, since they cannot be further instantiated.

  \[
  (?x owl:sameAs ?y) \rightarrow (?y owl:sameAs ?x)
  \]

The terminological rules are also referred as TBox rules, whereas the other two types constitute the ABox rules of DLEJena.

DLEJena implements the hybrid rules following an approach based on template rules. Each hybrid rule is expressed as a template rule that generates instantiated rules by removing all the TBox references from the condition of the hybrid rule and grounding the remaining unbound variables with actual TBox values. In that way, (a) more than one rule may be generated for a particular hybrid rule, and (b) the rule base of DLEJena is not predefined and it is formed at run-time, according to the schema constructs of the loaded ontology. Intuitively, a template rule can be viewed as a production rule that asserts dynamically other production rules. The instantiated rules are in fact exceptional rules, since they do not refer to TBox information in their body.

For example, the template rule for the inverse property relationship is:

\[
(?p owl:inverseOf ?q) \rightarrow [ (?x ?p ?y) \rightarrow (?y ?q ?x) ]
\]

If hasParent and hasChild are two inverse properties, then the above template rule is going to be instantiated with variables \(?p\) and \(?q\) bound to both the above properties and the following two (exceptional) ABox rules are going to be created:

\[
(?x hasParent ?y) \rightarrow (?y hasChild ?x) \\
(?x hasChild ?y) \rightarrow (?y hasParent ?x)
\]

#### 2.2.2.2 The DLEJena Architecture

The architecture of DLEJena is depicted in Figure 2 and comprises four modules: the Ontology Loader, the TBox Reasoner, the Template Processor and the ABox Reasoner. DLEJena makes use of the Jena API and therefore, it supports all the Jena-specific interfaces for conducting queries, e.g. SPARQL queries.

![Figure 2. The DLEJena architecture.](image)

### 3. REASONING INTEROPERABILITY

A core setback in agent interoperation is the variety in representation and reasoning. Despite KIF’s efforts [16], there is still no globally agreed knowledge representation and reasoning formalism for agents. For SW agents, on the other hand, we can safely assume that OWL will be the global knowledge exchange language. As for rule- and logic-based reasoning, there is a variety of proposals and standards ([10], [9], [54]). Thus, a key factor for the success of SW agents is reasoning task interoperability among
multiple, heterogeneous web entities exchanging rule bases to justify their positions.

Agents usually do not necessarily share a common rule or logic formalism. In fact, it will often be the case that two or more intercommunicating agents will ‘understand’ different (rule) languages. On the other hand, it would be unrealistic to attempt imposing specific logic formalisms in a rapidly changing world like the Web. Reasoning interoperability among agents can be often (but not always) achieved by translating the received rule set into the receiving agent’s formalism. This can only be accomplished, when the two agents use the same rule formalism with different syntax, or when the one formalism can be semantically translated into the other (e.g. translation between defeasible logic and DataLog rules [6]).

In this tutorial, we present a more viable approach, proposed in [32], which involves trusted, third-party reasoning services that will infer knowledge from an agent’s rule base and verify the results. More specifically, we will present the implementation of EMERALD, a framework for interoperating, knowledge-based IAs in the SW. A JADE MAS was extended with reasoning capabilities, provided as agents. Furthermore, the framework features a generic, reusable agent prototype for knowledge-customizable agents (KC-Agents) [31], consisted of an agent model (KC Model), a yellow pages service (Advanced Yellow Pages Service) and several external Java methods (Basic Java Library). Also, since the notion of trust is vital here, a reputation mechanism was integrated in the framework.

3.1 The EMERALD Framework

EMERALD is a common framework for interoperating knowledge-based intelligent agents in the SW, built on top of JADE [7]. EMERALD 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. The motivation behind this was to leverage the weaknesses in agent intercommunication. EMERALD supported, so far, the implementation of various applications, like brokering [32], price negotiations [31] and contract negotiations [30].

Figure 3 illustrates a general overview of EMERALD: Each human user controls a single all-around agent. Agents can intercommunicate, but do not necessarily share a common rule/logic formalism; therefore, it is vital for them to find a way to exchange their position arguments seamlessly. Our approach does not rely on translation between rule formalisms but on exchanging the rule base results. The receiving agent uses an external reasoning service to grasp the semantics of the rulebase, i.e. the set of rule base conclusions. In EMERALD, reasoning services are “wrapped” by an agent interface, called the Reasoner, allowing other IAs to contact them via ACL messages. Reasoners are, in essence, agents offering reasoning services to the rest of the agent community. Currently, the framework features a variety of widely diverse Reasoners (see section 3.2.2).

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 3 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.

Moreover, agents are knowledge-customizable, meaning that they are not confined in having their logics and strategies/policies hard-wired. Instead, they can be either generic or customizable; each agent contains a rule base that describes its knowledge of the environment, its behaviour pattern as well as its strategy/policy. By altering the rule base, the agent’s knowledge and/or behaviour will instantly be modified accordingly. Currently, EMERALD provides a knowledge-based agent module based on Jess [15] language, but it could be extended to provide a range of modules based on a variety of rule languages (i.e. Prolog/Prova, RuleML).

Overall, the goal is to apply as many standards as possible (ACL, RuleML, RDF/S, OWL), in order to encourage the application and development of the framework. In practice, the SW serves as the framework infrastructure.

3.2 Reasoners

3.2.1 Reasoner Functionality

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

The procedure is straightforward (Figure 3): 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.

A sample ACL message, based on FIPA [52] description, in the CLIPS-like syntax is displayed below:

\[
\text{(ACLMessage)} \\
\text{(communicative-act REQUEST)} \\
\text{(sender AgentA@xx:1099/JADE)} \\
\text{(receiver xx-Reasoner@xx:1099/JADE)} \\
\text{...}
\]
where AgentA sends to a Reasoner (xx-Reasoner) a RuleML file.

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. 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. 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.

3.2.2 Reasoning Services

EMERALD currently implements a number of Reasoner agents that offer reasoning services in two main formalisms: deductive and defeasible reasoning. Table 1 summarizes their main features.

<table>
<thead>
<tr>
<th>Reasoning engines supported by EMERALD</th>
<th>Type of Logic</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-DEVICE</td>
<td>deductive</td>
<td>RDF/CLIPS/RuleML</td>
</tr>
<tr>
<td>Prova</td>
<td>deductive</td>
<td>Prolog/Java</td>
</tr>
<tr>
<td>DR-DEVICE</td>
<td>defeasible</td>
<td>RDF/CLIPS/RuleML</td>
</tr>
<tr>
<td>SPINdle</td>
<td>defeasible</td>
<td>XML/Java</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order of Logic</th>
<th>Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-DEVICE</td>
<td>2nd order</td>
</tr>
<tr>
<td>Prova</td>
<td>1st order</td>
</tr>
<tr>
<td>DR-DEVICE</td>
<td>2nd order</td>
</tr>
<tr>
<td>SPINdle</td>
<td>1st order</td>
</tr>
</tbody>
</table>

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. EMERALD currently deploys two deductive reasoners, based on the logic programming paradigm: R-Reasoner and Prova-Reasoner, which deploy the R-DEVICE [5] and Prova rule engines [29], respectively.

Defeasible reasoning [43], on the other hand, constitutes a non-monotonic rule-based approach for efficient reasoning with incomplete and inconsistent information. EMERALD currently supports two defeasible reasoners: DR-Reasoner and SPINdle-Reasoner, which deploy DR-DEVICE [6] and SPINdle, respectively [33].

In the following subsection, we will describe in more detail R-DEVICE and DR-DEVICE; both systems were developed by the author. Furthermore, a brief insight into the fundamental elements of defeasible logics will be given, as well.

3.2.2.1 The R-DEVICE Reasoning Engine

R-DEVICE [5] 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 OPSS/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:

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

The above rule 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 with a semi-naive evaluation proof procedure and an OO syntax in the spirit of F-Logic [28]. 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.

3.2.2.2 The DR-DEVICE Reasoning Engine

DR-DEVICE [6] is a defeasible logic reasoner. Defeasible logic [43] is a simple and efficient rule based non-monotonic formalism, with a particular concern about efficiency and implementation. Over the year the logic has been developed and extended, and several variants have been proposed. The main intuition of the logic is to be able to derive “plausible” conclusions from partial and sometimes conflicting information. Conclusions are tentative conclusions, in the sense that a conclusion can be withdrawn when we have new pieces of information. When compared to more mainstream non-monotonic reasoning approaches, the main advantages of defeasible reasoning are enhanced representational capabilities and low computational complexity [35].

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 ($\succ$). Therefore, $D$ can be represented by the triple $(F, R, \succ)$. 
In defeasible logic, there are three distinct types of rules: strict rules, defeasible rules and defeaters. Strict rules \((A \rightarrow p)\) are interpreted in the typical sense: whenever the premises are indisputable, so is the conclusion. An example of a strict rule is: “Apartments are houses”, which, written formally, would become:

\[ r_1: \text{apartment}(X) \rightarrow \text{house}(X) \]

Defeasible rules are rules that can be defeated by contrary evidence and are denoted by \(A \not\rightarrow p\). An example of such a rule is “Any apartment is considered to be acceptable”, which becomes:

\[ r_2: \text{apartment}(X) \Rightarrow \text{acceptable}(X) \]

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:

\[ r_3: \text{pets}(X), \text{gardenSize}(X,Y), Y>0 \in \text{acceptable}(X) \]

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, the following rule:

\[ r_4: \text{pets}(X) \Rightarrow \neg\text{acceptable}(X) \]

Finally, the superiority relationship among the rule set \(R\) is an acyclic relation \(>\) on \(R\). For example, given the defeasible rules \(r_2\) and \(r_4\), no conclusive decision can be made about whether the apartment is acceptable or not, because rules \(r_2\) and \(r_4\) contradict each other. But if a superiority relation \(>\) with \(r_4 > r_2\) is introduced, then \(r_4\) overrides \(r_2\) and we can indeed conclude that the apartment is considered unacceptable. In this case rule \(r_4\) is called superior to \(r_2\) and \(r_2\) inferior to \(r_4\).

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 for a given apartment is acceptable or not, because rules

\[ \text{offer}(X) \]

overrides \(\text{pets}(X)\) and \(\text{gardenSize}(X,Y)\), \(Y>0 \in \text{acceptable}(X)\) 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, the following rule:

\[ \text{pets}(X) \Rightarrow \neg\text{acceptable}(X) \]

For completeness, we also include the representation of rule \(r_3\) in the CLIPS-based syntax, in order to demonstrate rule superiority and negation:

\[ \text{defeasible rule r4} \]

\[ \Rightarrow\]

\[ \text{apartment (name ?X) (pets "no")} \]

\[ \text{(not (acceptable (name ?X)))} \]

### 3.3 The KC-AGENTS Prototypes

EMERALD also supports a customizable, knowledge-based agent model, the KC-Agents prototypes. KC-Agents are customizable agents equipped with a KB and a Jess rule engine. The KB contains the agent’s knowledge (facts) and behavior pattern (production rules). KC-Agents are described by an abstract specification portrayed below that contains facts and rules; the generic rule format is represented by: \(\text{result} \Leftarrow \text{rule (preconditions)}\). The agent’s internal knowledge is essentially a set \(F\) of facts that consists of subset \(F_u\) of user-defined facts and subset \(F_e\) of environment-asserted facts:

\[ F_u \equiv \{f_{u1}, f_{u2}, ..., f_{uq}\}, F_e \equiv \{f_{e1}, f_{e2}, ..., f_{em}\}, F \equiv F_u \cup F_e \]

Agent behavior is a set \(P\) of potential actions, expressed as Jess production rules. \(P\) consists of rules that derive new facts by inserting them into the KB (subset \(A\)) and rules that lead to the execution of a special action (subset \(S\)). Special actions can either refer to agent communication (subset \(C\)) or Java calls (subset \(J\)):

\[ P \equiv A \cup S, S \equiv C \cup J \]

\[ A \equiv \{a_j \Leftarrow \neg \text{a}(f_{u1}, f_{u2}, ..., f_{uq}) \} \]

\[ C \equiv \{c_j \Leftarrow \text{c}(f_1, f_2, ..., f_n)\} \]

\[ J \equiv \{j \Leftarrow \text{j}(f_1, f_2, ..., f_n)\} \]

ACLMessage is a Jess template for defining ACL messages and JavaMethod is a user-defined Java method. The communication rule syntax specification is:

\[ \text{(defrule Communication \_ Rule} \]

\[ \text{... rule preconditions} \]

\[ \Rightarrow (\text{ACLMessage (communicative-act ?a) (sender ?s) (receiver ?r) (content ?n)}) \]

where communicative-act, sender, receiver and content are four template parameters of ACLMessage, according to Fipa. On the other hand, user-defined Java methods can be called inside Jess rules to perform a specialized action, like processing specialized file content. A generic syntax specification is:

\[ \text{(defrule JavaMethod \_ Rule} \]

\[ \Rightarrow \]

\[ \text{(JavaMethod (method-fun ?x) (arg1 ?a, arg2 ?b))} \]

\[ \text{(result \_)} \]

\[ \text{endrule} \]

\[ \text{endrule} \]
... rule preconditions

\[\begin{align*}
(b \, ?x \, (\text{new\_java\_class\_name})) \\
(b \, ?y \, (\text{?x\_java\_method\_name}\ ?a))
\end{align*}\]

where \(?x\) is bound to a new instance of a specific Java class, \(?a\) is the list of arguments required by the specific class method and \(?y\) is the returned result.

### 3.4 Use Case: A Brokering Scenario

In this subsection we present the implementation in EMERALD [32] of a defeasible reasoning-based brokering scenario that was adopted from [1]. 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.

#### 3.4.1 Scenario Overview

The scenario is carried out in eight distinct steps, as shown in Figure 4. Carlo’s agent retrieves the corresponding apartment schema, 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. These requirements are expressed in defeasible logic, in the DR-DEVICE RuleML-like syntax. All RuleML and RDF files exchanged in this scenario can be found at the DR-DEVICE site.

![Figure 4. The steps of the brokering scenario.](image)

The broker, on the other hand, has a list of all available apartments, along with their specifications (stored as an RDF database), 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 URL of the RDF document containing the list of available apartments, and stands by for the list of proper apartments (step 2).

Then, DR-Reasoner launches DR-DEVICE, which processes the above data and returns an RDF document, containing the apartments that fulfill all requirements. 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. 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 fulfill his “special” rules (step 5).

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. 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.

#### 3.4.2 KC-Agent Specifications

Following the generic specification for KC-Agents, the customer agent’s description contains a fact, ruleml_path, which is part of its internal knowledge and represents the rulebase URL. Moreover, due to the dynamic environment, a new fact with the agent name is added to the working memory. Agent behavior is represented by rules; two of these are “request” and “read”; the former is used for communication and the latter for Java method calls. Both rules require a single precondition each.

\[F^\text{read} \equiv \{\text{ruleml\_path} \}, \quad F^\text{read} \equiv \{\text{agent\_name}\}\]

\[C^\text{read} \equiv \{(\text{ACLMessage (communicate-act REQUEST)}) \}
\]

\[J^\text{read} \equiv \{\text{rule\_base\_string} \leftarrow \}
\]

The broker agent’s description contains facts and rules: url represents its internal knowledge and stands for the URL of the RDF document containing all apartments, while reasoner_name (i.e. the Reasoner’s name) is added by the environment and rules “request” and “read” comprise part of the agent’s behavior.

---

\[ F^\text{break}_u \equiv \{ \text{url} \}, \quad F^\text{break}_e \equiv \{ \text{reasoner}_\text{name} \} \]
\[ C^\text{break} \equiv \{ (\text{ACLMessage}) \}
\text{(communicative-act REQUEST)}
\text{(sender Broker) (receiver reasoner}_\text{name} \text{)}
\text{(content "request")}
\leftarrow \text{ request (reasoner}_\text{name} \text{)} \]
\[ J^\text{break} \equiv \{ \text{rule_base_string} \leftarrow \text{(bind ((new java_class) read url))} \}\]

4. CONCLUSIONS
This tutorial discussed about issues, technologies and tools that concern the way that the Semantic Web affects knowledge and information interchange among intelligent agents in multi-agent systems, as well as reasoning interoperability. First, the tutorial discussed how semantic web rules and ontologies interact with each other in order to be used as the agent’s internal knowledge base for environment awareness and decision making. Then, interoperability between reasoning systems for agents was discussed and the EMERALD framework was presented. The issues involved in all the previous discussion have been exemplified using tools for semantic web reasoning in multi-agents systems, mostly implemented by teams led by the author.

5. ACKNOWLEDGMENTS
This work is partially supported by the Greek R&D General Secretariat through a bilateral Greek-Romanian project.

6. REFERENCES


