Integration between Agents and Remote Ontologies for the Use of Content on the Semantic Web

Felipe Demarchi, Elder Rizzon Santos and Ricardo Azambuja Silveira
Department of Informatics and Statistics, Federal University of Santa Catarina, Florianópolis, Brazil

Keywords: Intelligent Agents, Semantic Web, Jason Interpreter.

Abstract: The Semantic Web proposes a structure of significant content for Web pages that is used in knowledge bases and developed from ontologies, that have recently come to coexist on the Web. There are studies to allow agents to navigate through these knowledge bases in search of answers to queries. This work proposes the adaptation of a well-known agent structure, named Jason, in order to allow the agent access to ontologies available on the Web. In this context, efforts have been made to perform the integration of agents with ontologies, most of which allow the knowledge of the agent to be based on a local ontology. However, applying the ability to use semantic data available on the Web to a consolidated belief-desire-intention (BDI) agent structure is a subject that still needs to be explored. Therefore, this work proposes changes in the implementation of the Jason interpreter that would allow agents to access ontologies available on the Web to perform the update of their belief base based on significant content. As validation, a case study of an educational quiz is presented that uses this information to formulate the questions and validate the answers obtained.

1 INTRODUCTION

The emergence of the Semantic Web has aroused interest in research involving computational intelligence.

According to Berners-Lee, Hendler and Lassila (2001), the Semantic Web aims to bring a meaningful content structure to Web pages, allowing virtual agents to move between these pages performing specific tasks for users. In order to do this, it is necessary to use knowledge representation from the various ontologies that are available on the Web. The great power offered by the Semantic Web will be accessible when agents can collect the information available in the various bases of knowledge representation, process this information and share the results with other agents.

For the definition of agents, we consider the notation presented by Wooldridge and Jennings (1995), according to which an agent consists of a system that is situated in some environment and is capable of performing autonomous actions in this environment to reach its objectives. It has the properties of autonomy, social ability, reactivity and proactivity.

BDI is based on a human behavior model developed by philosophers, having its origin in the theory of human practical reasoning, with a focus mainly on intentions in the reasoning practice developed by Bratman (1987). Wooldridge (2002) explained that the BDI system consists of the process of deciding, moment by moment, which action to take to reach a certain goal. Therefore, Bordini et al. (2007) defined that these computer programs have a computation analogous to beliefs, desires and intentions.

Another concept necessary for the context of the Semantic Web refers to ontologies. Gruber (1995) defined an ontology as an explicit specification of a conceptualization, understood as a simplified and abstract vision of the world that is meant to represent it for some purpose. Antoniou and Van Harmelen (2008) pointed out that an ontology consists of a finite list of terms. Furthermore, the relationship between these terms, which defines important concepts, is formed by the classes and objects of the domain.

Many approaches have been developed to allow agents to use ontologies as a knowledge base. Dikenelli et al. (2005), Moreira et al. (2006), Klapiscak and Bordini (2009), Mascardi et al.

Demarchi, F., Santos, E. and Silveira, R.
Integration Between Agents and Remote Ontologies for the Use of Content on the Semantic Web.
DOI: 10.5220/0006718701250132
In *Proceedings of the 10th International Conference on Agents and Artificial Intelligence (ICAART 2018) - Volume 1*, pages 125-132
ISBN: 978-989-758-275-2

(2011), Campos (2014) and Freitas et al. (2015) all formed proposals that allow the integration between agents and ontologies as a knowledge base. However, these proposals seek to present the agent's integration with local ontologies, not exploring the possibility of accessing remote ontologies available on the Semantic Web.

In view of this, the current research presents the integration of a consolidated BDI agent structure, named Jason, with remote ontologies made available from databases such as DBPedia. As validation, a case study is presented that consists of an educational quiz related to geography topics, which will use the proposed agent model to define the questions and answers from searches of these knowledge bases available on the Web.

This paper is organized as follows. Related studies are described in Section 2. Section 3 presents the changes applied to the Jason interpreter. In Section 4, a case study is described that validates the proposed model. Finally, Section 5 presents the conclusions and future work.

2 RELATED WORK

In this section, we present the notable studies that perform an integration between agents and ontologies, and focus on those that use ontologies as the agents' knowledge base.

Dikenelli, Erdur and Gumus (2005) propose the SEAGENT, a model that allows agents to have a local ontology as internal knowledge, allowing the communication between these agents. Since the agents can have heterogeneous ontologies as a knowledge base, it used ontology matching between the ontologies of the two agents, allowing agents with heterogeneous ontologies to communicate with each other.

Moreira et al. (2006) presented a theoretical model of a BDI agent-oriented programming language called AgentSpeak-DL, an extension using descriptive logic and ontologies of the AgentSpeak language, which uses predicate logic. To do this, it incorporated ontological knowledge with the agent, presenting the necessary changes in language semantics to allow execution based on these ontologies.

Klapiscak and Bordini (2009) describe JASDL (Jason AgentSpeak-DescriptionLogic). This study used the Jason interpreter to implement the theoretical proposal presented by Moreira et al. (2006), demonstrating the changes made to allow Jason to use ontological reasoning to update his belief base and retrieve relevant plans.

Mascardi et al. (2011) presented CooL-AgentSpeak, an extension of the AgentSpeak-DL language that allows alignment between the local heterogeneous ontologies present in different agents. It makes use of an agent with the alignment capability called Ontology Agent, which is consulted whenever it is necessary to perform an alignment between the ontologies of two agents.

Campos (2014) introduced PySA, a Phyton BDI agent implementation that defines URIs (Uniform Resource Identifiers) as agent beliefs that point to online data available on the Semantic Web, more specifically in DBPedia.

Freitas et al. (2015) proposed an approach that allows the interaction of agents and ontologies using a coded layer based on CArtAgO. In this approach, any agent-oriented language with support for this artifact can use this implementation to perform the integration between agents and local ontologies. One of the main contributions of this work is to allow an agent to have access to more than one ontology as a knowledge base.

In analysing the above-mentioned works, it is clear that much effort is being applied to research concerning the integration between agents and ontologies, with the aim of contributing to the research related to the Semantic Web. Although, many research finding can support the use of ontologies as the agents' knowledge base, as well as, in some cases, support the communication among the agents working in multi-agent systems, among the related works found, the work of Campos (2014) is the only one that integrates the agents' knowledge base and a remotely available ontology. But this model is presented through a proper ad hoc implementation of the agent. That is, without applying any consolidated or well-known framework for BDI Agents.

3 MODIFICATION PROPOSED IN THE AGENT MODEL

In order to apply the concepts of the Semantic Web to a consolidated agent model, we chose to use the architecture and the reasoning cycle of the Jason interpreter.

According to Bornidi et al. (2007), this interpreter uses ten steps for the execution of the reasoning cycle of an agent. The first four correspond to obtaining information for the belief base, which can happen from communication with

other agents or based on the perception of the environment.

The other steps work with the selection of events and plans that allow them to reach the objectives of the agent and initially consists of the selection of an event. All relevant plans are retrieved for this event and the applicable plans defined. Finally, one of these plans is selected and one of its intentions is obtained.

This work proposes changes in the implementation of the Jason interpreter, in order to allow him to use existing knowledge in the Semantic Web to feed his belief base. However, no modification was performed on the model structure, as the execution of the reasoning cycles were maintained exactly as proposed by Bordini et al. (2007).

In the sequence, we present the modifications made to the implementation of the Jason interpreter. They consist of the class responsible for allowing the execution of queries to remote ontologies and the creation of three internal events that allow the agents to make use of these queries.

In order to prepare a case study that involves creating an educational quiz related to geography topics, the definitions of some internal events are directed towards this end. However, the Jason Interpreter allows the creation of new internal events, thus allowing other definitions to be implemented.

3.1 Queries to Remote Knowledge Bases

In order to allow the Jason interpreter to support queries to remotely available knowledge bases through the SPARQL query language (i.e. DBPedia), a class called SparqlSearch was added to the implementation, which is solely responsible for performing these queries. To enable this support, the Jena framework was used.

This class consists of the definition of the searchDbpedia method, which receives a SparqlObject parameter and returns a list of objects. SparqlObject was an object created only to represent a triple, having the attributes called URI, Predicate and Object. The return consists of a list of type Object by the fact that the query can return resources or literals.

There are two options for creating the SPARQL query. The first one is based on a URI and a predicate to find the corresponding objects, while the second is based on the predicate and the object to obtain the corresponding URI. Therefore, taking the predicate and the URI, or the object, it is possible to query DBPedia using this class. Figure 1 shows the part of the code responsible for defining the SPARQL query.

```
String querySearch = "";
if (sparqlObject.getUri() != null
        && !sparqlObject.getUri().isEmpty()) {
    querySearch = sparqlObject.getUri() + " " +
                  sparqlObject.getPredicate() +
                  " ?result .";
} else if (sparqlObject.getObject() != null) {
    querySearch = "?result " +
                  sparqlObject.getPredicate() + " ";
    if (sparqlObject.getClass().equals(String.class)) {
        querySearch += (String) sparqlObject.getObject();
    } else if (sparqlObject.getClass().equals(Integer.class)) {
        querySearch += (int) sparqlObject.getObject();
    }
    querySearch += " .";
}

String query = PREFIXES +
        "SELECT ?result WHERE {"
        + "SERVICE <http://DBpedia.org/sparql> {"
        + querySearch
        + "}"
        + "}";
```

Figure 1: Code demonstrating the SPARQL query.

In the query variable definition, the PREFIXES constant contains all the prefixes required to perform a SPARQL query on DBPedia. The return of the query brings a list of results, for which it would check whether there are resources or literals to add to the method's return list.

3.2 Searchdbpedia Internal Event

To allow a Jason agent to check the validity of a predicate and a URI or Object, an internal event named searchdbpedia was added to the implementation.

This event receives two terms as arguments to check whether it is a valid query or not. This is necessary because you can request a query based on a URI or Object and a predicate that are not related, or based on an incorrect URI, so this internal event is necessary to perform this validation. With this, the agent can define rules for which event or plan to execute based on the return of the execution of this internal event. As an example, the definition of contexts using this internal event is presented below.

```
+!search : .searchdbpedia(
"<http://dbpedia.org/resource/Brazil>",
"dbo:country") <- ...

+!search : not .searchdbpedia(
"<http://dbpedia.org/resource/Brazil>",
"dbo:country") <- ...
```

In this example, if the internal searchdbpedia event returns some SPARQL query result using the

URI for Brazil next to the dbo:country predicate, then the first event will be selected for execution, otherwise it will be the second.

The implementation of the execute method of this internal event gets the terms passed as arguments, in this case the URI and the predicate. They are converted to the type String and perform the query using the class SparqlSearch. Figure 2 shows the part of the code referring to this step.

```
if (sparqlObject.checkUriAndPredicate()) {
    List<Object> results =
        SparqlSearch.create().
            searchDbpedia(sparqlObject);
    if (!results.isEmpty()) {
        return true;
    }
}
return false;
```

Figure 2: The code related to the internal searchdbpedia event.

After validating that a URI and a predicate have been defined for the query, a list of results is obtained from the execution of the SPARQL query to be performed using the SparqlSearch class. If any result is obtained, it returns true, otherwise it returns false.

3.3 Checkuri Internal Event

When performing SPARQL queries, it is necessary to allow the agent to use the values obtained from the query to feed its belief base. For this, the checkuri internal event was defined, which aims to verify if the query returned is a resource. If true, it will allow the agent to add this resource to its belief base.

For the implementation of this internal event, the SPARQL query is performed as previously shown, checking if the object obtained from the URI and the predicate informed match a URI that points to another DBPedia entity. Figure 3 demonstrates the implementation responsible for this event.

```
List<Object> results = SparqlSearch.create().
        searchDbpedia(sparqlObject);
for (Object result : results) {
    if (result.getClass().equals(ResourceImpl.class)) {
        Resource resource = (Resource) result;
        Term t = new StringTermImpl("<"+resource.getURI()+">");
        return un.unifies(args[3], t);
    }
}
```

Figure 3: The code related to the checkuri internal event.

This event receives three terms as arguments. The first two refer to the URI and the predicate, whereas the third consists of a term to be unified with the URI received as a result of the query. In this implementation, if the query returns more than one URI as a result, only the first one will be unified with the argument. Below is an example of using this internal event from a Jason agent.

```
+!check : .checkuri(
"<http://dbpedia.org/resource/Florianópolis>", "dbo:country", X ) <- ...
```

In this example, the internal event receives the URI referring to the city of Florianópolis and the dbo:country predicate for the SPARQL query. With this, the result will be the URI referring to the entity Brazil, which will be unified with the variable X.

3.4 Checkanswer Internal Event

The first two internal events presented, searchdbpedia and checkuri, correspond to the definition of an event pattern necessary for agent integration with remote ontologies. The checkanswer internal event was defined to fit the context of the case study used by this work, which consists of an educational quiz. However, it also emphasized the fact that by inserting the SparqlSearch class into the Jason interpreter, internal events can be added to the Jason project in order to address specific situations.

For the context of the case study, it is necessary to verify if the response given by the student matches the result obtained from the SPARQL query. In this case, if the result is a literal, it is converted to its respective data type and the comparison is performed. If it is a resource, it will be necessary to perform a new query to obtain the name for this resource and for this, a query is performed using the foaf:name predicate.

This internal event receives three terms as arguments, where the first refers to the response sent by the student and the last two refer to the URI and the predicate needed to conduct the query. Figure 4 shows the code referring to the case where the result obtained is a resource. In this way, a new query is performed to obtain a literal referring to the name of the resource in order to perform the validation of the response.

Based on the result obtained in the second query, it is observed if the answer matches the name referring to this resource.

```
if (result.getClass().
        equals(ResourceImpl.class)) {

    Resource resource = (Resource) result;
    SparqlObject sparqlForName =
            new SparqlObject("<"+resource.getURI()+">",
                             "foaf:name");
    List<Object> resourceName =
            SparqlSearch.create().
            searchDbpedia(sparqlForName);

    for (Object name : resourceName) {
        Literal lName = (Literal) name;
        String sName = lName.getString();
        String[] splitName = sName.split("@");
        if (sAnswer.equalsIgnoreCase(splitName[0])) {
            return true;
        }
    }

}
```

Figure 4: The code related to the response case being a resource in the checkanswer internal event.

If the result of the query is a literal, then the second query is not required. Only the conversion of the literal to its specific data type is required in order to allow comparison with the response, as shown in Figure 5.

```
if (result.getClass().equals(LiteralImpl.class)) {
Literal lResult = (Literal) result;
if (lResult.getValue().getClass().
        equals(Integer.class)) {
    try {
        int answerConverted = Integer.parseInt(sAnswer);
        if (lResult.getInt() == answerConverted) {
            return true;
        }
    } catch (NumberFormatException nfe) {
        return false;
    }
} else if (lResult.getValue().getClass().
        equals(String.class)) {
    String[] split = lResult.getString().split("@");
    if (sAnswer.equalsIgnoreCase(split[0])) {
        return true;
    }
} else if (lResult.getValue().getClass().
        equals(XSDDateTime.class)) {
    XSDDateTime dateResult =
            (XSDDateTime) lResult.getValue();
    String splitDate[] = sAnswer.split("-");
    try {
        if (dateResult.getYears() ==
            Integer.parseInt(splitDate[0].trim()) &&
            dateResult.getMonths() ==
            Integer.parseInt(splitDate[1].trim()) &&
            dateResult.getDays() ==
            Integer.parseInt(splitDate[2].trim())) {
            return true;
        }
    } catch (NumberFormatException nfe) {
        return false;
    }
}
```

Figure 5: The code related to the response case being a resource in the checkanswer internal event.

In this case, it is initially necessary to convert the literal to its specific data type and then perform the comparison. As a return, it will be obtained true or false, referencing whether the answer given by the student is correct or not. The call to this event from a Jason agent is shown below.

```
+!answer : .checkanswer( “Brazil”,
“<http://dbpedia.org/resource/Florianóp
olis>”, “dbo:country”) <- ...

+!answer : not .checkanswer( “Brazil”,
“<http://dbpedia.org/resource/Florianóp
olis>”, “dbo:country”) <- ...
```

The first case concerns the student providing the correct answer, while the second concerns the student’s misunderstanding.

4 CASE STUDY

To exemplify the execution of agents with the ability to access data available in the Semantic Web, a case study will be demonstrated consisting of a quiz related to geography subjects.

To do so, it will be necessary to work with two agents: (1) the question agent (QA) that is responsible for formulating the questions and (2) the answer agent (AA) that is responsible for receiving the questions and sending the response.

The QA agent will begin its process with a belief called uri, which corresponds to a list that will initially contain only the resource of DBPedia about the city of Florianópolis, and the number of questions that will be responsible for managing the questions to be asked based on each URI. In addition, it will initially have the objective of running the event called generatedQuestions, which is responsible for formulating the questions. The initial state of the agent is represented as follows:

```
uri([<”http://dbpedia.org/resource/Flor
ianópolis”>]) .
num_of_questions(0) .

!generateQuestions .
```

As predicates were used for the realization of the questions, some relationships identified in the entities were pre-established, which can be observed in Figure 6. In this case, some predicates result in an empty value, while others point to literals or other resources. These will be analysed based on the internal events added to the Jason interpreter.

The generateQuestions event that is executed initially is responsible for initializing the generation

of the questions. Several plans are defined that unify with this event, having as a definition the number of the question verified in the context to define which of the plans will be applicable.

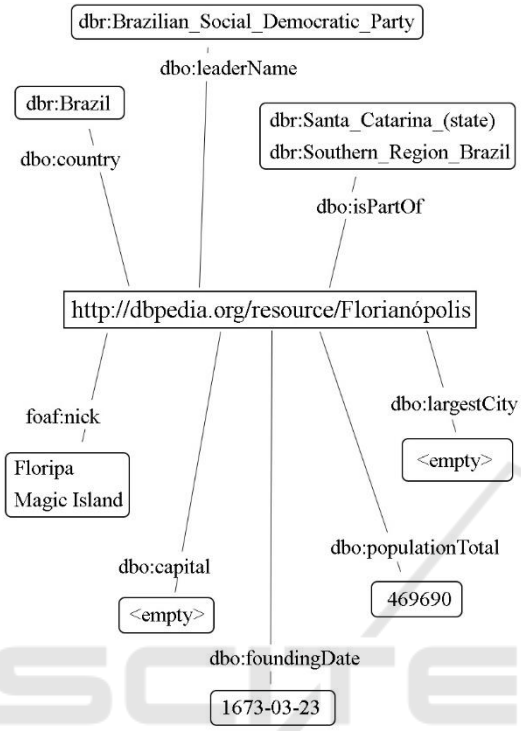

Figure 6: Properties related to the URI that reference Florianópolis.

To exemplify the step for the generateQuestions event, three of the possible plans for this event are as follows:

```
+!generateQuestions : num_of_questions(
X ) & X = 0 <- !question("where",
"dbo:country", "located"); -
+num_of_questions(X+1).
+!generateQuestions : num_of_questions(
X ) & X = 1 <- !question("when",
"dbo:founding", "located"); -
+num_of_questions(X+1).
+!generateQuestions : num_of_questions(
X ) & X = 2 <- !removeUri; -
+num_of_questions(0);!generateQuestions
.
```

The first two check whether the value corresponding to the belief num_of_questions corresponds to a predetermined value to then execute the event called question, and increment the value referring to the belief num_of_questions by 1. The last event checks whether a question has been reached for a given URI. When this occurs, the event named removeUri, executes and sets the value corresponding to the belief num_of_questions to zero. The generateQuestions event is called again, which will begin to formulate questions to a new URI.

For the event called removeUri, the agent separates the list of URIs that has as belief in Head and Tail. This belief is redefined with the value referring to Tail, that is, without the first element for which they have already been performed questions.

The event called question validate if the relation of a URI with the predicate informed as argument corresponds to a SPARQL query valid to DBPedia. This should occur if the result of the query is not empty. To do so, the searchdbpedia internal event is used, and two question events are defined for the agent. For one case the query is possible and for the other case it is not, as presented in the following codes:

```
+!question(Type, Predicate, Word) :
uri(L) & L = [H|T] & .searchdbpedia(H,
Predicate) <- .send(answerAgent,
achieve, question(H, Type, Predicate,
Word).

+!question(Type, Predicate, Word) :
uri(L) & L = [H|T] & not .searchdbpedia
(H, Predicate) & num_of_questions(X) <-
-+num_of_questions(X+1);
!generateQuestions .
```

In the first situation, where the return of the search dbpedia internal event is true, you get the first element of the uri belief list. Then sends this information to the AnswerAgent agent, triggering your event.

In the second case, if there is no query return for the URI and the predicate in question, one is added to the num_of_questions belief of the agent and the generateQuestions event is executed again.

Agent AA has the event named question, which in this example checks whether this event was triggered from the QA agent. If so, the question is presented and response is returned to the QA agent, triggering the answer event. The response is sent with the URI and the predicate in question, so the QA agent can perform the query to validate the response, as shown in the following code.

```
+!question(Uri, Type, Predicate, Word)
[source(questionAgent)] <- .print(Type,
" is ", Uri, " ", Word); .send(
questionAgent, achieve,
answer("Brazil", Uri, Predicate).
```

After this step, agent QA will execute the answer event, which has two definitions - one in case the answer is correct and another in case it is incorrect. To perform the validation of the response, it uses the internal event created called checkanswer, which will return true or false.

```
+!answer(Answer, Uri, Predicate) :
.checkanswer(Answer, Uri, Predicate) <-
.print("Congratulations"); !verifyUri(
Uri, Predicate, Answer);
!generateQuestions.

+!answer(Answer, Uri, Predicate) :
.checkanswer(Answer, Uri, Predicate) <-
.print("Wrong answer!");
!generateQuestions.
```

If the response is incorrect, only a message is displayed and the process for generating a new question is initiated. If correct, a congratulatory message is displayed and the verifyUri event is executed.

```
+!verifyUri(Uri, Predicate) :
.checkuri(Uri, Predicate, X) <-
!addUri(X) .
```

This event is responsible for verifying that the result obtained from the combination of the URI and the predicate corresponds to a URI that points to another DBPedia entity. To do this, it uses the internal event added to Jason called checkuri, which performs this verification, and if it identifies that it corresponds to a URI, it then unifies this value to variable X. In the sequence, the event named addUri is executed, which adds the URI obtained at the end of the list of URIs that the agent has as belief for later formulation of questions.

Thus, during the execution of a cycle of formulating questions based on a given URI, new URIs were obtained and added as agent beliefs, which maintained a relation referring to the content addressed by the initial URI.

5 CONCLUSIONS

This research addressed the use of the concept of Semantic Web together with the implementation of the Jason interpreter. The principles of a consolidated BDI agent model were maintained, enabling agents implemented from this tool to have the ability to access data available in remote ontologies for the production and updating of beliefs.

In order to do so, the creation and modification of internal events of the Jason interpreter were proposed. The implementation of a class to be used to intermediate the SPARQL queries to remote bases, more specifically DBPedia, were also proposed. Thus, from a URI that the agent has as a belief, it is possible to expand its knowledge based on the relations obtained from this URI and from predicates, always maintaining content coherence.

In order to validate the proposed model, a case study was presented for the creation of an educational quiz about geography, in which the agents exclusively use information available in remote ontologies to determine the questions related to the context. With this, it was possible to observe that from an initial belief, the agent manages to produce and expand the bank of questions based on entities that relate to the initial belief, maintaining the coherence of the content.

The main contribution of this study was to provide modifications in a well-known BDI agent model, in this case the Jason interpreter, allowing agents to integrate with ontologies available on the Web. It is important to point out that some of the internal events presented in this work refer to the context of a virtual learning environment, for the production of an educational quiz. Following the same context, and using the main class of the proposal, called SearchDbpedia, it is possible to define new internal events in order to meet other contexts.

About the related works described above, it is possible to observe that the sequence of research presented by Moreira et al. (2011), Klapiscak and Bordini (2009), Mascardi et al. (2011) and Freitas et al. (2015) propose changes in consolidated BDI architectures in order to allow access to ontologies, however, do not define the possibility of access to remote ontologies and, in some cases, they use ontology matching algorithms to allow the communication between agents. By proposing a model in which the agents have the capability to update their beliefs based on remote ontologies, we claim that is possible that the group of agents has a common-sense knowledge base. In this case, the case study uses DBPedia, which eliminates the need for ontology matching algorithms.

The research work proposed by Campos (2014), presents an agent model that supports beliefs revision according to the knowledge obtained from remote ontologies. This model uses an ad hoc implementation to perform the validation of the

proposed schema. Therefore the model not completely matches the basic principles of a BDI architecture. Based on this, we have chosen to use the Jason interpreter, which is a consolidated BDI agent model, without changing its reasoning cycle, but just working with internal events that allow agents access to remote ontologies. Thus our model implementation can be used for purposes other than the case study used to validate the proposal.

For future work, we proposed allowing an agent to perform SPARQL queries on more than one web-based knowledge base, such as Wikidata and GeoNames. In addition, better standardization of internal events that adapt to more generic situations can be established.

REFERENCES

Antoniou, G. and Van Harmelen, F. (2008). *A Semantic Web Primer*. (2nd ed.). London: The MIT Press.

Berners-Lee, T., Hendler, J. and Lassila, O. (2001). *The Semantic Web*. Scientific American, May 2001. Available from: https://pdfs.semanticscholar.org/566c/1c6bd366b4c9e07fc37eb372771690d5ba31.pdf [Accessed 15/10/17].

Bordini, R. H., Hübner, J. F. and Wooldrige, M (2007). *Programming Multi-Agent Systems in AgentSpeak using Jason*. England: Wiley.

Bratman, M. (1897). *Intention, plans, and Practical Reason*. Cambridge, Mass, Harvard University Press.

Campos, D. (2014). *Representação de Dados Semânticos em Agentes BDI*. Dissertation (Msc.), Federal University of Santa Catarina.

Dikenelli, O., Erdur, R. C. and Gumus O. (2005). SEAGENT: A Platform for Developing Semantic Web Based Multi Agent Systems. In: *Proceedings of the Fourth International Joint Conference on Antonomous agent and Multiagent Systems*. Utrecht: ACM, pp. 1271-1272.

Freitas, A., Panisson, A. R., Hilgert, L., Meneguzzi, F., Vieira, R. and Bordini, R. H. (2015). Integrating Ontologies with Multi-Agent Systems through CArtAgO Artifacts. In: *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*. Singapore: IEEE, pp. 143-150.

Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing? *International Journal of Human-Computer Studies*. 43(5-6), pp. 907-928.

Klapiscak, T. and Bordini, R. H. (2008). JASDL: A Practical Programming Approach Combining Agent and Semantic Web Technologies. In: *6th International Workshop on Declarative Agent Languages and Technologies*. Estoril: Springer, pp. 91-110.

Mascardi, V., Ancona, D., Bordini, R. H. and Ricci, A. (2011). CooL-AgentSpeak: Enhancing AgentSpeak-DL Agents with Plan Exchange and Ontology Services. In: *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*. Lyon: IEEE, pp. 109-116.

Moreira, A. F., Vieira, R., Bordini, R. H. and Hübner, J. F. (2005). Agent-Oriented Programming with Underlying Ontological Reasoning. In: *Third International Workshop on Declarative Agent Languages and Technologies*. Utrecht: Springer, pp. 155-170.

Wooldrige, M. (2002). Intelligent Agents: The Key Concept. In: *Multi-Agent Systems and Aplications II*. Berlin: Springer, pp. 3-43.

Wooldrige, M. and Jennings, N. R. (1995). Intelligent agents: theory and practice. *The Knowledge Engeneering Review*. 10(2), pp. 115-152.