Agent-Based Simulation with Beliefs and SPARQL-based Ask-Reply Communication

Ion Mircea Diaconescu[1] and Gerd Wagner[1]

[1]Chair of Internet Technology
Institute of Informatics
Brandenburg University of Technology, Germany
{M.Diaconescu, G.Wagner}@tu-cottbus.de

Abstract. We present the result of extending an agent-based simulation framework by adding a full-fledged model of beliefs and by supporting ask-reply communication with the help of the W3C RDF query language *SPARQL*. Beliefs are the core component of any cognitive agent architecture. They are also the basis of ask-reply communication between agents, which allows social learning. Our approach supports the conceptual distinctions between facts and beliefs, and between sincere answers and lies.

Keywords: cognitive agent simulation, beliefs, reasoning, RDF, SPARQL.

1 Introduction and Motivation

While allowing to model many complex simulation scenarios, today's agent-based simulation systems, such as SESAM [3], REPAST [4] or NetLogo [8], do not offer much support for modeling beliefs and ask-reply communication based on beliefs. The situation is different in the area of agent programming languages. E.g., in [1] a solution for automated belief revision in the language AgentSpeak [6] is presented. But unlike cognitive agent simulation systems, agent programming languages are not concerned with the important conceptual distinction between *facts* and *beliefs*.

In this paper we present a solution for modeling the beliefs of an agent about its environment and about itself, and for supporting belief-based ask-reply communications between agents with the help of the W3C RDF [2] query language *SPARQL* [5]. Our solution is obtained as an extension of the open source *Agent-Object-Relationship (AOR) Simulation* framework[1], which is an ontologically well-founded agent-based discrete event simulation framework with a high-level rule-based simulation language, *AORSL*, and an abstract simulator architecture and execution model.

AOR Simulation supports the distinction between facts and beliefs by maintaining both the objective and the subjective state of an agent in parallel using

[1] Available from http://AOR-Simulation.org.

the two classes *AgentObject* and *AgentSubject*. A fact about an agent is represented by means of a slot of the corresponding instance of the AgentObject class, while a belief about the agent (called a *self-belief* in AORSL) is represented by means of a slot of the corresponding instance of the AgentSubject class.

Beliefs about objects in the environment (including other agents) cannot be represented by simple *property-value slots*, like self-beliefs, but need to be represented by *object-property-value triples*, also called "subject-predicate-object" triples in the jargon of RDF. Therefore, we have extended AORSL by adding a construct for defining *belief entity types* as part of the definition of an agent type. A belief entity type defines a number of belief properties for expressing belief triples about an object of some type.

The concept of belief entity types allows to represent all kinds of beliefs of an agent about its environment, no matter which vocabulary (or ontology) the agent is using. In this way, agents could either use a shared vocabulary, or they could use their own private vocabularies, which would have to be mapped to each other for successful communication. However, in this paper, we do not consider the problems of private vocabularies and ontology mapping. For simplicity, we assume that all agents are using a shared vocabulary, including shared identifiers for all objects of the simulation scenario.

2 Introduction to AOR Simulation

AOR Simulation was proposed in [9]. It supports both basic discrete event simulations without agents and complex agent-based simulations with (possibly distorted) perceptions and (possibly false) beliefs. A simulation scenario is expressed with the help of the XML-based *AOR Simulation Language (AORSL)*. The scenario is then translated to Java source code, compiled to Java byte code and finally executed, as indicated in Figure 1.

Fig. 1. From AORSL to Java byte code.

A **simulation scenario** consists of a *simulation model*, an *initial state* definition and zero or more *view* definitions.

A **simulation model** consists of: (1) an optional *space model* (needed for physical objects/agents); (2) a set of *entity types*, including different categories of event, message, object and agent types; and (3) a set of *environment rules*, which define *causality laws* governing the environmental state changes.

An **entity type** is defined by means of a set of *properties* and a set of *functions*. There are two kinds of properties: attributes and reference properties.

Attributes are properties whose range is a data type; *reference properties* are properties whose range is another entity type.

An **agent type** is defined by means of: (1) a set of (objective) *properties*; (2) a set of (subjective) *self-belief properties*; (3) a set of (subjective) *belief entity types*; and (4) a set of *agent rules*, which define the agent's reactive behavior in response to events.

A **space model** is characterized by the parameters: (1) dimension (1D, 2D or 3D); (2) discrete/continuous; (3) geometry (Euclidean or Toroidal); and (4) space limits (xMax, yMax, zMax).

The upper level **ontological categories** of AOR Simulation are objects (including agents, physical objects and physical agents), messages and events, as depicted in Figure 2. Notice that according to this upper-level ontology of AOR Simulation, agents are special objects. For simplicity it is common, though, to say 'object' instead of the unambiguous term *non-agentive object*.

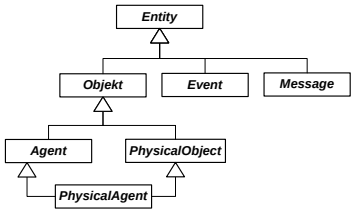

Fig. 2. Upper-level ontological categories.

An elaborate ontology of **event types**, shown in Figure 3, has proven to be fundamental in AOR Simulation. Internal events are those events that happen *'in the mind'* of the agent. For modeling distorted perceptions, both a perception event type and the corresponding actual perception event type can be defined and related with each other via actual perception mapping rules.

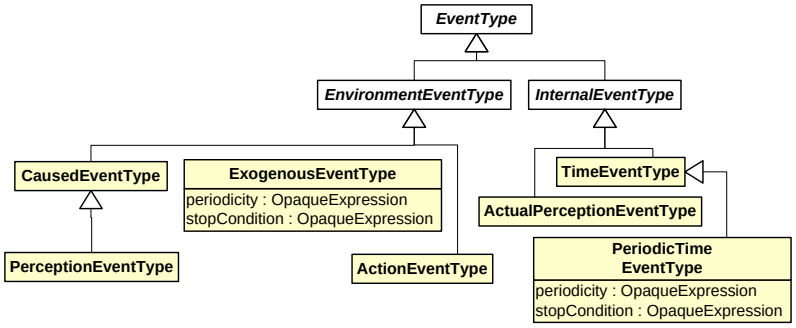

Fig. 3. Categories of event types.

Both the behavior of the environment (its causality laws) and the behavior of agents are modeled with the help of **rules**, thus supporting high-level declarative behavior modeling. An **environment rule** is a 5-tuple $\langle EvtT,\ Var,\ Cond,\ UpdExpr,\ ResEvtExpr \rangle$, where: (1) *EvtT* denotes the type of event that triggers the rule; (2) *Var* is a set of variable declarations, such that each variable is bound either to a specific object or to a set of objects; (3) *Cond* is a logical condition formula, allowing for variables; (4) *UpdExpr* specifies an update of the environment state; and (5) *ResEvtExpr* is a list of resulting events, which will be created when the rule is fired.

3 Modeling Beliefs

3.1 Self-Beliefs

When defining an agent type, we can not only define its (objective) *attributes*, which are used to express fact statements about agents of that type, but we can also define its *self-belief attributes*, which are used to express belief statements of agents of that type about themselves. The following definition of an agent type *Foo* contains both kinds of attributes:

```
<AgentType name="Foo">
 <Attribute name="position" type="Float"/>
 <Attribute name="velocity" type="Float"/>
 <SelfBeliefAttribute name="position" type="Float"/>
 <SelfBeliefAttribute name="myFavoriteNumber" type="Integer"/>
</AgentType>
```

The agent type *Foo* is then implemented with the help of two classes, as shown in Figure 4:

1. The class *FooAgentObject* representing the objective state of Foo agents with the help of the attributes x and v (for *velocity*).
2. The class *FooAgentSubject* representing the subjective state of Foo agents with the help of the attributes x and *myFavoriteNumber*.

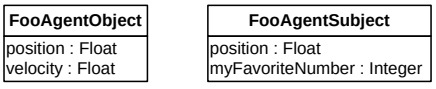

Fig. 4. An agent is divided into an object and a subject

Notice that according with this definition, Foo agents have a *position* and a *velocity*. They also have a self-belief about their position and another self-belief about about their favorite number, but not about their velocity. A self-belief attribute that corresponds to an objective attribute need not have the same name. Instead of *position*, a Foo agent type definition could use another name, say *myPosition*, for expressing beliefs about their position.

In general, a fact about an agent is represented by means of a slot of the corresponding instance of the AgentObject class, while a self-belief about the agent is represented by means of a slot of the corresponding instance of the AgentSubject class.

3.2 Belief Entity Types

For defining the types of beliefs an agent may have about the entities in its environment, *belief properties* applying to all entities of some type are grouped with the help of *belief entity types*. For instance, a belief entity type (`Castle`) may be defined for the agent type (`Knight`) in order to allow beliefs about the location of castles:

```
<PhysicalAgentType name="Knight">
 <BeliefEntityType name="Castle">
   <BeliefAttribute name="x" type="Integer"/>
   <BeliefAttribute name="y" type="Integer"/>
 </BeliefEntityType>
</PhysicalAgentType>
```

In AORSL, any entity type is defined to be a class in the sense of the UML, as shown in Figure 5. Therefore, any entity type has a number of properties. Belief entity types specialize entity types, since they have a number of belief properties, which specialize properties by imposing the constraint that their domain (the entity type to which they belong) must be a belief entity type component of an agent type (called 'believer type' in the metamodel shown in Figure 5). Self-belief properties specialize belief properties by imposing the constraint that their domain is the believer type (that is, they are properties of instances of the believer type).

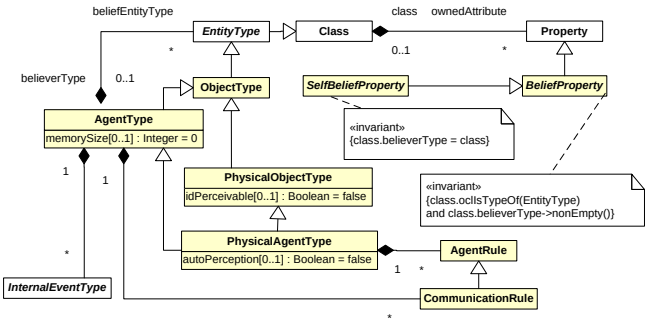

Fig. 5. Modeling agents with beliefs

3.3 Facts and Beliefs Are Represented by Triples

More precisely speaking, we do not deal with 'facts' and 'beliefs', but with *atomic fact statements* and *atomic belief statements*, each of them having the form of

an *object-property-value triple*. For instance, the atomic fact statements that the positions of the Foo agents with identifiers "007" and "008" are given by x = 347.2, resp. x = 12.7, is expressed by the triples

(It's a fact that) 007 position 347.2
(It's a fact that) 008 position 12.7

while the atomic belief statement of agent "007" that its position is given by x = 346.9 is expressed by the triple

(Agent 007 beliefs that) 007 position 346.9

and the atomic belief statement of agent "007" that the position of agent "008" is given by x = 13.1 is expressed by the triple

(Agent 007 beliefs that) 008 position 13.1

In standard predicate logic syntax, such a triple corresponds to an atomic sentence where the property of the triple statement would be used as a predicate, and the object identifier and the properties value would be the arguments of this predicate, resulting in the expression:

(It's a fact that) position(007, 347.2)

As discussed below, AORSL supports the use of the W3C RDF query language SPARQL for expressing queries about the beliefs of other agents in Ask messages. RDF defines a language for expressing triples in multiple vocabularies.

3.4 Perfect Information Agents

By default, if no self-belief properties and no belief entity types are defined for an agent type, agents of that type possess *perfect information*. These agent types have all their objective properties duplicated as self-belief properties and all self-belief slots have the same values as the corresponding objective slots.

3.5 Discrepancies between Fact Statements and Belief Statements

In general, we can have various types of discrepancies between fact and belief statements. The first issue is the possibility to use different languages to express statements about the same fact. Assuming that the same languages (i.e. the same names for entity types and properties and the same identifiers for individuals) are used, we still have the possibility of discrepancies between a fact statement and a corresponding belief statement with respect to the actual and the believed value of a property.

There are several types of possible discrepancies arising from different vocabularies being used. Agents may use different names for entity types and/or properties, and they may use different identifiers for individuals. There are also the issues of *partiality* and *non-correspondence*. Partiality refers to the possibility that not all 'real' entity types and properties (as defined objectively for the

environment of a simulation model) have a corresponding name in the vocabulary of an agent. Non-correspondence refers to the possibility that some of the entity type and/or property names used by an agent do not correspond to a real entity type or property.

AOR Simulation allows modeling of all these kinds of discrepancies between fact and belief statements. However, we are still investigating the required inference capabilities of agents for being able to map the vocabularies of other agents to their own when they communicate with each other.

3.6 Belief Handling

An agent may create new beliefs, or it may change or destroy existing beliefs. The way how an agent manages its beliefs is defined with the help of *agent rules*.

For instance, as in the scenario presented in next section, knights have beliefs about the castle and about magic objects found on the map. Since the prince may have already discovered some magic objects, when he asks a knight about the next moving direction towards the nearest magic object, he will first inform the knight about any magic objects already found. Moreover, the prince has to create new beliefs about any magic object found. The following example is an excerpt form an agent rule of the agent type Knight and a prince rule showing how beliefs may be created and destroyed.

```
<!-- the Prince creates beliefs about a discovered magic object -->
<UpdateSubjectiveStateExpr>
  <CreateBeliefEntity beliefEntityType="MagicObjectBelief">
    <BeliefEntityId language="Java">
      e.getPerceivedPhysicalObject().getId()
    </BeliefEntityId>
  </CreateBeliefEntity>
</UpdateSubjectiveStateExpr>

<!-- the Knight destroy beliefs about already discovered magic objects -->
<UpdateSubjectiveStateExpr>
  <DestroyBeliefEntity>
    <BeliefEntityRef beliefEntityType="MagicObjectBelief" language="Java">
      this.getBeliefEntityById(((Ask)e.getMessage()).getFoundMagicObjectId())
    </BeliefEntityRef>
  </DestroyBeliefEntity>
</UpdateSubjectiveStateExpr>
```

The communication between the prince and knight agent is message based. The prince sends his request via an *Ask* message and the knight replies by sending a *Reply* message. The generic Ask/Reply message types may be adapted for a specific problem domain, such as in this example, the message contains specific information about the magic object near the SPARQL query. The following example defines an *Ask* message type used by the prince when he asks a knight about the nearest magic object. The property `foundMagicObjectId` refers to the ID of an already discovered magic object:

```
<MessageType name="AskAboutMagicObject">
  <Attribute type="String" name="queryLanguage"/>
  <Attribute type="String" name="queryString"/>
```

```
  <Attribute type="Integer" name="foundMagicObjectId"/>
</MessageType>
```

4 The Simulation Scenario Test Case

A simulation scenario based on a *'quest game'* is used to exemplify and test capabilities discussed in this paper. This scenario was mainly used as a test case during this research. The used story is simple: a *Prince* wants to rescue the *Princess* kidnaped by the *Evil Demon*. First, the *Prince* has to improve his power by finding some *magic objects* on the map. Until the *Prince* has at least the same power level as the *Demon*, he will ask any found *Knight* about the next moving direction towards the *magic object*. When the power level is greater than the *Demon*'s power, he will start to ask about the direction towards the *Castle*.

A set of rules and axioms defines the simulation: (1) the prince has beliefs about the demon's power level; (2) while the demon's power is constant, the prince's one may be increased by finding some magic objects; (3) a constant number of magic objects providing additional power are available on the map; (4) knights are randomly distributed and they can provide the next moving direction towards the castle or the nearest magic object; (5) the map is a grid space, and the prince can move only one cell during each simulation step, in one of the four directions: E, W, S or N; (6) the prince sees a knight, a magic object, the castle or the demon when he enters in the same cell where this is placed; (7) the castle and the demon are in the same cell; (8) the prince rescue the princes only if at the moment when he discover the castle his power level is greater than the demon's one. (9) the prince has basic learning capabilities. He tries to find *the best* moving direction when no knight is found in a cell.

The following scenarios are possible: (i) the castle is found before the prince has the requested power level; (ii) the prince does not find the castle before the stimulation steps are finished; (iii) the prince finds the castle, defeat the demon, rescue the princess and marry her.

5 An RDF-based Representation for Agent Beliefs

In this section a solution to represent the beliefs of an agent as an RDF graph (a conjunction of triple statements) is presented. The main purpose of the RDF representation is to have a standard representation of beliefs, together with a standard query language (SPARQL). The main advantages of this approach are that it allows to:

- use any SPARQL (or other RDF query answering) engine;
- use RDF-based reasoning engines, such as Jena Rules [7] or ERDF [10] as a middleware layer between an agent's beliefs and the query level;
- express Semantic-Web-based simulation scenarios (e.g. social networks simulations).

An AOR simulation model defines a `baseURI` attribute, having as value an URI. This is used as a base URI to define RDF triples. Moreover, each `Entity` type has a unique `ID` during its live cycle. Two types of RDF triples are defined for beliefs:

1. type definition triples, (b rdf:type T), where:
   - $b = [baseURI] + "/" + [AgentType] + "/" + [BeliefType] + "/" + [ID]$;
   - $T = [baseURI] + "/" + [AgentType] + "/" + [BeliefType]$;
2. property value triples, (b prop val), where:
   - $b = [baseURI] + "/" + [AgentType] + "/" + [BeliefType] + "/" + [ID]$;
   - $prop = [baseURI] + "/" + [AgentType] + "/" + [BeliefType] + "/" + [prop\_name]$;
   - $val = Literal \quad OR \quad TypedLiteral \quad OR \quad URIRef.$

For example, having the `Castle{id=501, x=12, y=15}` beliefs for a `Knight` agent, and `baseURI = 'http://aor.org/KK'`, the following RDF triples are generated :

```
http://aor.org/KK/Knight/Castle/501 rdf:type http://example.com/KK/Knight/Castle;
                http://aor.org/KK/Knight/Castle/x "12"^^xs:integer;
                http://aor.org/KK/Knight/Castle/x "15"^^xs:integer.
```

6 Querying Beliefs with SPARQL

In our simulation scenario, the `Prince` has to ask `Knights` about the moving direction towards the nearest *magic object* or towards the `Castle`. The communication is made via Ask/Reply messages. The *request message* encapsulates the SPARQL query and the ID of the already found magic object. The *response message* contains the moving direction calculated by the *knight*.

```
<OutMessageEventExpr messageType="AskAboutMagicObject">
  <ReceiverIdRef language="Java">
    e.getPerceivedPhysicalObjectIdRef()
  </ReceiverIdRef>
  <Slot xsi:type="aors:SimpleSlot" property="queryLanguage" value="SPARQL"/>
  <Slot xsi:type="aors:SimpleSlot" property="queryString"
        value="SELECT ?x ?y WHERE {?c rdf:type :MagicObjectBelief;:x ?x;:y ?y.}"/>
  <Slot xsi:type="aors:SimpleSlot" property="foundMagicObjectId">
    <ValueExpr language="Java">prince.getBeliefEntityByType(0).getId()</ValueExpr>
  </Slot>
</OutMessageEventExpr>
```

The default (and built-in) namespace (expressed as `:`) represents the value of `baseURI` attribute and the agent type (e.g. `http://aor.org/KK/Knight/`).

The `Knight` replies with an *Reply* message containing the moving direction towards the nearest magic object.

```
<OutMessageEventExpr messageType="ReplyAboutTheMagicObject">
  <ReceiverIdRef language="Java">e.getSenderIdRef()</ReceiverIdRef>
  <Slot xsi:type="aors:OpaqueExprSlot" property="messageReference">
    <ValueExpr language="Java">(int)e.getMessage().getId()</ValueExpr>
  </Slot>
  <Slot xsi:type="aors:OpaqueExprSlot" property="answer">
    <ValueExpr language="Java">
```

```
      knight.computeAnswer(((Ask)e.getMessage()).getQueryString(),
                           ((Ask)e.getMessage()).getFoundMagicObjectId())
    </ValueExpr>
  </Slot>
</OutMessageEventExpr>
```

A query can be executed by calling the `executeQuery(queryString)` method. As result, a `List` containing `HashMap` instances, is returned. Each element of the list represents a solution of the query. Concrete values of a solution are extracted by using the corresponding keys. The variable names used in queries are the needed keys. For instance, `x` and `y` variable names are used in the above SPARQL query, and values bounded to these variables are extracted by using `x` and `y` keys. Finally, these values have to be converted to the appropriate Java type (e.g. `int` for the above example).

7 Results of Running the Simulation Test Case

In this paper, all examples are based on the *Knights&Knaves* simulation scenario We considered a benchmark for it, and two cases are defined: (1) *Knight* agents capable of helping the *prince* are used; (2) the *prince* has no external help and it uses it's own *basic* capabilities to find magic objects and the princess. Figure 6 shows a screen-shot taken during a simulation. The blue square represents the *Castle*, yellow squares are *Magic Objects*, green circles symbolizes *Knights* and the red circle represents the *Prince*.

A set of 13 tests, each of them consisting in 100 simulations of 1000 steps each, was made. The number of *Knight* agents was increased with 25 for each new test. In Figure 7A a statistic of the case when `Knight` agents are available during the simulation is provided, in contrast with the case from Figure 7B where `Knight` agents are not used. It is obvious the difference of game wins between the two cases. Moreover, it is obvious that the information exchange between agents can be very important for scenarios where agents have to accomplish a defined task.

One may argue that, different approaches and simulation engines of this scenario may offer the same (or even more improved) results. This may be true, but using a standard beliefs representation (RDF) and a standard query language (SPARQL) may have the advantage of expressing complex belief models and complex queries without using complicate technologies which may require more learning effort. Moreover, nowadays a number of SPARQL query engines implementations (e.g. ARQ[2]) and RDF based information representation (e.g. RDF models in Jena[3]) are available as open source projects.

8 Conclusions and future work

We have presented a solution for dealing with a full-fledged model of beliefs in agent-based simulation. Moreover, we have shown how the RDF query language

[2] ARQ Web Page - http://jena.sourceforge.net/ARQ/
[3] Jena Web Page - http://jena.sourceforge.net/

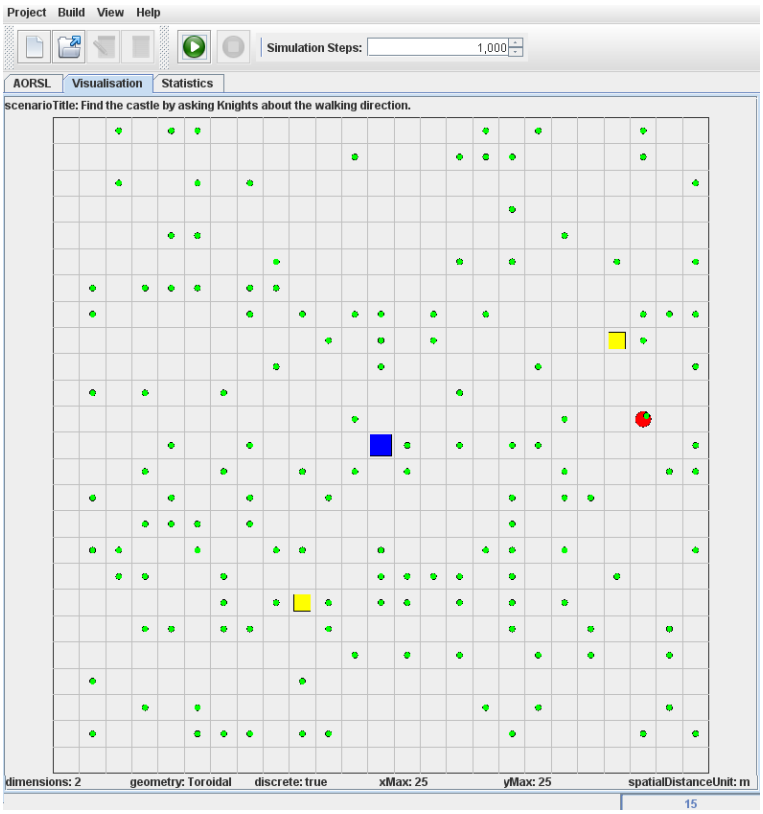

Fig. 6. Knights&Knaves visualisation

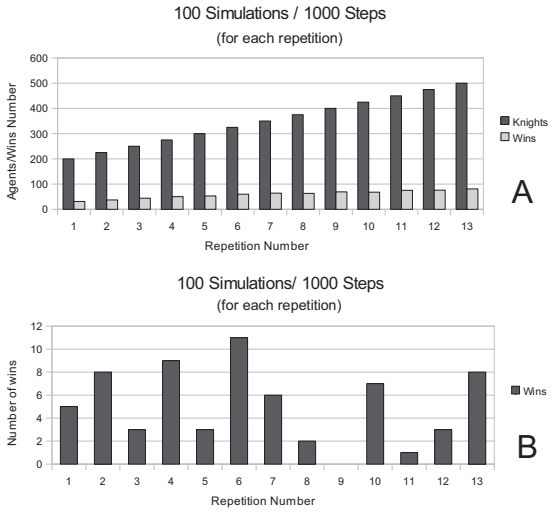

Fig. 7. Knights&Knaves Statistics - (A) With Knights; (B) Without Knights

SPARQL can be used for implementing a model of ask-reply communication between cognitive agents. A simulation scenario dealing with these new capabilities has been described and analyzed. In future work we will turn the Java-based communication code into more high-level constructs that extend the current version (0.6) of our simulation language AORSL. Another important step in our research is to integrate an inference engine as a middle layer between the RDF triples representation and SPARQL queries. This requires to define a representation of production and/or derivation rules as a further extension of AORSL.

Acknowledgments: Thanks to Jens Werner for his help provided to implement these improvements, and to Dr. Adrian Giurca for his helpful advice and support.

References

1. Natasha Alechina, Rafael H. Bordini, Jomi F. Hbner, Mark Jago, and Brian Logan. Automating belief revision for agentspeak. In M. Beladoni and U.Endriss, editors, *Proceedings of the Fourth International Workshop on Declarative Agent Languages and Technologies (DALT 2006), held with AAMAS 2006, 8th May*, pages 1–16. Springer-Verlag, 2006.
2. Klyne G. and Caroll J.J. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation 10 February 2004. http://www.w3.org/TR/rdf-concepts/.
3. Franziska Kluegl and Frank Puppe. The Multi-Agent Simulation Environment SeSAm. In *Proceedings of Workshop "Simulation in Knowledge-based Systems", 1998 (Report tr-ri-98-194, Reihe Informatik, Universitt Paderborn)*, 1998.
4. Michael J. North, Tom Howe, Nick Collier, and Jerry R. Vos. The Repast Simphony Development Environment. In *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms, ANL/DIS-06-5*, pages 159–166. Argonne National Laboratory and The University of Chicago, 2005.
5. Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/, November 2007.
6. A.S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J.W. Perram, editors, *Agents Breaking Away*, volume 1038 of *LNAI*, pages 42–55. Springer-Verlag, 1996.
7. Dave Reynolds. Jena Rules experiences and implications for rule use cases. In *W3C Workshop on Rule Languages for Interoperability*, 2005.
8. S. Tisue and U. Wilensky. NetLogo: Design and implementation of a multi-agent modeling environment. In *Proceedings of 8th Annual Swarm Users/Researchers Meeting*, Ann Arbor, MI, 9–11 May 2004.
9. Gerd Wagner. *Agent-Oriented Information Systems*, volume 3030 of *LNAI*, chapter AOR Modelling and Simulation - Towards a General Architecture for Agent-Based Discrete Event Simulation, pages 174–188. Springer-Verlag, 2004.
10. Gerd Wagner, Adrian Giurca, Ion-Mircea Diaconescu, Grigoris Antoniou, Anastasia Analyti, and Carlos Viegas Damasio. Reasoning on the Web with Open and Closed Predicates. In Jos de Bruijn, Stijn Heymans, David Pearce, Axel Polleres, and Edna Ruckhaus, editors, *Proceedings of the 3rd International Workshop on Applications of Logic Programming to the (Semantic) Web and Web Services (ALP-SWS2008)*, Udine, Italy, December 2008. CEUR Workshop Proceedings.