Inserting "brains" into software agents – preliminary considerations

Maria Ganzha[1,3], Mariusz Marek Mesjasz[1], Marcin Paprzycki[1], and Moussa Ouedraogo[2]

[1] Systems Research Institute Polish Academy of Sciences, Warsaw, Poland,
`<firstname>.<lastname>@ibspan.waw.pl`
[2] CRP Henri Tudor, Luxembourg,
`<firstname>.<lastname>@tudor.lu`
[3] Institute of Informatics, University of Gdansk, Gdansk, Poland

Abstract. Software agents are often seen as "intelligent, autonomous software components." Interestingly, the question of efficient implementation of "intelligence" remains open. In this paper we discuss, in some details, the process of implementing software agents with "brains." In the context of an agent system supporting decisions of glider pilots, we consider native implementation of "intelligent" behaviors, rule based engines, and semantic data processing. Based on the analysis of the state-of-the-art in these areas, we present a novel approach combining rule based engines, semantic data processing and software agents.

1 Introduction

One of the interesting issues in design and implementation of agent systems is: how to make agents "intelligent." Note that, very often, software agents are conceptualized as "intelligent, autonomous software components, which interact with each other in order to achieve goals for benefits of their users" [29, 31]. However, as illustrated below, it is not easy to find an agent platform, where a *robust* (and *flexxible*) method for making agents intelligent is available.

Separately, a number of rule-based expert systems have been developed. Typically, they are based on the RETE pattern matching algorithm [24]. Implementation of RETE has been completed, among others, in C, C++ and Java, and provide user interfaces and/or definition of the API.

Finally, since the 1980's (see, [22, 30]) an explosion of research in ontologies and semantic technologies ensued. This has culminated, over the last 10 years, in rapid sprawl of ontology management tools (e.g. Protege [17]) and reasoners (with Hermit [8], Pellet [16] and Fact++ [7] being the most popular).

Let us now consider an agent-based support system for glider pilots (for more details, see [23, 27]). The idea is to aid the glider pilot in events that may occur during a flight. For instance, to detect life-treating situations, warn the pilot and, autonomously, inform the ground station. The pilot-supporting agent runs on a tablet or a smart-phone. For the system we have selected the Android OS [1] and the JadeAndroid add-on [12] (which allows running full Jade agent container on

Android devices). The next step to be undertaken is to introduce "reasoning capabilities" to the *GliderAgent* agents. This provides the conceptual backdrop for the explorations reported below.

Note that we are interested in practical aspects of implementing agent systems (see, also, [26, 28]). Furthermore, we believe in open source solutions and reuse of state-of-the-art software. This gives the methodological foundation of our work. Therefore, in the next section, we summarize approaches to implementing "brains" of software agents. Specifically, we consider "native" approaches (within agent platforms), rule-based expert systems, and semantic technologies. We follow with details of the implementation of the selected approach.

2 Introducing intelligence into software agents

2.1 Native approaches

Large number of agent platforms have been written in different languages. Since we are interested in agent systems running on Android devices, we discuss those written in the Java. However, to the best of our knowledge, discussion presented here applies to majority of agent platforms in use today.

Jade [12] (Java Agent DEvelopment Framework; version 4.3.2; 2014-03-28) is written entirely in Java, with the lead developer being Telecom Italia. Jade developers provided add-ons to run it on mobile devices. JadeAndroid is an official add-on that allows to run Jade agents on Android OS. Jade agents store their knowledge in a form of *Java classes*. As a result, after Java classes are compiled, they *cannot* be changed without altering the *entire* application. Thus, an agent has to be recompiled, each time when a change is introduced into its "knowledge." Therefore, it is not trivial to introduce changes to a running Jade agent system (without taking the system down and restarting it; see [25]). Obviously, in the GliderAgent system (as well as majority of real-world multi-agent systems), a more flexible approach is desired. Namely, an intelligent agent should be able to add, modify or delete a "behavior" (related to a "knowledge fragment") on-demand (or, at least, without the need of restarting the system).

JASON [11] (version 1.4.0a; 2013-12-17) is an interpreter for an extended version of the AgentSpeak (based on the Belief Desire Intention (BDI) paradigm). It is available as an Open Source software (GNU LGPL license). JASON agents are written in AgentSpeak. Since no official version of JASON for mobile devices exists, it is unclear how easy it would be to run it on smart-phones. Furthermore, JASON requires additional infrastructure (e.g., Jade) to run it distributed over a network. Thus, one may assume that JASON *and* an additional agent platform have *both* to be ported to the Android API. This doubles the amount of required programming work, and may also result in "doubling" resource consumption.

In the JadeX agent platform [10], "intelligence" is facilitated in the form of XML-based BDI rules. However, in our context, JadeX has three disadvantages. (1) It has somewhat irregular development cycle. (2) It is evolving from a high-level BDI-agent platform to the agent-as-component model, which has much smaller granularity and, as a result, it is not clear what, if any, will be the role of the XML-demarcated BDI rules in its future releases. (3) It remains unclear how easy / difficult it would be to run JadeX on mobile devices.

JACK agents (developed by the AOS Group; [9]) use the JACK Agent Language, a super-set of Java. JACK compiler converts source code of JACK agents into pure Java. Therefore, JACK agent platform has the same limitations as Jade. Namely, agent code has to be recompiled each time when an update is required. Furthermore, as of now, JACK does not provide official support for the Android OS (or any other mobile device). Note that it may be quite difficult to port JACK to Android because JACK agents have to be translated twice: (1) from the JACK Agent Language to the Java programming language, and (2) from the Java programming language to the Android API. Since (1) is done by the JACK compiler, developers have limited control over this process. Consequently, there is a substantial risk that the resulting Java code will not work with the Android OS. Furthermore, any change in the agent system (e.g. change in agent "knowledge") would have to be done within the JACK subsystem first, and then moved to the actual system. Therefore, perspective of using JACK on mobile devices (or in a mixed mobile-non-mobile environment) is not appealing.

2.2 Rule-based expert systems for agent systems

Let us now consider the possibility of combining a Rule-based Expert System (RES) with Jade agents running on the Android OS (however, this discussion generalizes also to other agent systems and other mobile devices). First, note that Java is not natively compatible with Android. While having the same syntax and similar interfaces, there are key differences between the Java API and the Android API. (1) Android does not use the Java Virtual Machine. Instead, it uses the Dalvik or the ART (starting from Android 4.4); two runtime environments written and maintained by Google. (2) Not all Java packages were included in the Android API; e.g. the *javax* package (Swing, XML libraries, etc.) is missing, and has to be replaced by the Android's native classes. Therefore, the RES should:

- Be an open source project (with license that permits modifications). Due to the differences between the Java API and the Android API, the RES (and its dependencies) will have to be ported (and possibly modified). Moreover, parts of the RES may need to be replaced by the Android native libraries to work and/or to achieve better performance.
- Be an "active project." Note that many open source projects are being developed by "independent" / academic teams. Therefore, the risk of selecting an unfinished or obsoleted project is high, and should be avoided. Moreover, dealing with older versions of Java may increase the amount of work needed to make the project operational.

Finally, an ideal RES should have a small number of dependencies and a simple structure. This should minimize rewriting the required libraries. Moreover, complex projects are difficult to understand and modify, without introducing errors. However, these considerations are secondary.

To compare the existing RESs, we used data found at [13]. After refreshing the information, and selecting RES's written in Java, we have summarized it in Table 2.2. For each RES (row in the table) we include: name, latest stable version (and release date), license type and website.

It is easy to notice that only three RESs (Roolie, Drools, OpenL Tablets) are of interest in the current context. The remaining ones either are significantly outdated, or have multiple licenses (depending on the type of the project / the character of the licensee), which unnecessarily complicates their use "across the board." Therefore, we provide more information about these three.

– Roolie [18] (version 1.1, 2013-12-13) is an extremely simple RES. It chains user-defined rules (stored in XML files) to create more complex rules. Due to its size (only few kilobytes) and lack of dependencies, it seems that it could be adapted to the Android API. Unfortunately, it does not include a pattern matching algorithm. Thus, an additional algorithm (e.g. RETE) would have to be implemented. Moreover, Roolie is poorly documented.
– Drools [5] (version 6.0.0, 2013-12-20) is a forward and backward chaining rule engine. From the current version, it provides its own, enhanced, implementations of the RETE algorithm, called PHREAKY. Rules are stored in a Drools native language. The entire project is very well documented. Unfortunately, while very robust, it has many external dependencies. This would make Drools difficult to port to the Android API. Furthermore, its size and scope poses question about its usability on resource-limited mobile devices.
– OpenL Tablets [14] (version 5.12, 2014-04-21) is a full-blown expert system. Its rules and policies exist as an unstructured set of Excel and Word documents. This makes rules easier to understand and change by non-technical users. Unfortunately, proprietary data formats (Excel and Word) are not well-suited for an open source type system. Furthermore, the Android API may require extra libraries to open / manage them.

2.3 Semantic technologies for agent systems

Finally, to implement "brains" of software agents one could use semantic technologies. Here, facts are stored inside ontologies in a form of triples (subject, predicate, object), applications can use a reasoner to infer logical consequences from them. Statements within an ontology can be divided into: (i) a set of facts (A-box), and (ii) conceptualization associated with them (T-box). T-box defines a schema in terms of controlled vocabularies (for example, a set of classes, properties and axioms), while A-box contains T-box-compliant facts. Combination of the A-box and the T-box makes up a knowledge base. Interestingly, the Drools developers officially stated that, in the future, they will try to bring OWL Lite to Drools. This shows growing interest in combining these two technologies. Since, we are interested in open source solutions, we have considered Apache Jena and the OWL API.

Apache Jena [3] (Jena, version 2.11.1, 2013-09-18) is a, well-documented, open source framework for building semantic web and Linked Data applications.

Rule-based Engine	Developer(s)	Stable release	License	Website
Drools	Red Hat	6.0.0 / 2013-12-20	ASL 2	`http://drools.jboss.org/`
DTRules	Andreas Viklund	4.3 / 2011-07-05	ASL	`http://dtrules.com/`
Hammurapi Rules	Hammurapi Group	5.7.0 / 2009-01-11	LGPL	`http://www.hammurapi.biz/`
Jena rule-based reasoner	Apache Software Foundation	2.11.1, 2013-09-18	ASL 2	`http://jena.apache.org/`
JEOPS	Ravi Tangirala, Bob Schlicher, Jeff Carter	2.2 / 2003-09-29	LGPL	`http://sourceforge.net/projects/jeops/`
JLisa	Mike Beedle	0.04 / 2003-11-20	GPL	`http://jlisa.sourceforge.net/`
JRuleEngine	Mauro Carniel	1.3 / 2008-04-16	LGPL	`http://jruleengine.sourceforge.net/`
Mandarax	Jens Dietrich, Jochen Hiller, Alex Kozlenkov	1.1.0 / 2011-01-26	LGPL	`https://code.google.com/p/mandarax/`
Open Lexicon	OpenLexicon.org	1.0.4 / 2007-01-12	ASL	`http://openlexicon.org`
OpenL Tablets	OpenL Tablets	5.12 / 2014-04-21	LGPL	`http://openl-tablets.sourceforge.net/`
OpenRules	OpenRules, Inc.	6.3.1 / 2014-05-18	GPL / Non-GPL Licenses for Commercial Projects	`http://openrules.com`
Roolie	Ryan Kennedy	1.1, 2013-12-13	LGPL	`http://roolie.sourceforge.net/`
SweetRules	MIT Sloan and DAML	2.1 / 2005-04-25	LGPL	`http://sweetrules.projects.semwebcentral.org/`
TermWare	GradSoft	2.3.3 / 2011-06-16	Other	`http://www.gradsoft.ua/products/termware\_eng.html`

It provides an API to extract data from and write to RDF, RDFS and OWL graphs. Graphs are loaded from: (i) file system, (ii) database, or (iii) the web (via URLs) and represented as abstract structures called "models." The Jena works with a) RDF, b) OWL, and c) triple store. It includes popular semantic reasoners: Fact++ [7], Pellet [16] and HermiT [8]. Furthermore, it provides its own implementation of the SPARQL 1.1 engine (the AQR). Note that, even though Jena was not designed as a rule-based engine, it implements the RETE algorithm in a general purpose rule-based reasoner. This reasoner is used for updating the loaded ontologies, when a certain rule is met. There existed two community projects aiming at running Jena on Android. (1) The Androjena [2] project supported only a subset of the Jena features and was discontinued in 2010. (2) The Apache Jena on Android [4] project tried to fully integrate Jena (with all its features) with the Android OS. Unfortunately, the latest, stable version of Apache Jena on Android was released for the outdated Jena, in version 2.7.3 released on August 7, 2012.

The OWL API [15] (version 3.50, 2014-04-07) is a Java API (and reference implementation) for creating, manipulating and serialising OWL Ontologies. It supports OWL 2.0 and offers an API to inference engines and ontology validation. Similarly to Jena, the OWL API provides interfaces for FaCT++, HermiT, Pellet and Racer (but they are not build-in). Furthermore, it features: (1) an API for OWL 2.0, (2) parsers and writers for RDF/XML, OWL/XML, OWL Functional Syntax, Turtle, KRSS, and OBO Flat. To the best of our knowledge, are no ("official" or community-driven) projects intended to port the OWL API to the Android OS (or other mobile OS).

3 Implementing agents with "brains" on mobile devices – proposed approach

We have considered different approaches to infuse software agents with intelligence. Despite the fact that native methods provided by agent platforms are sufficient for many scenarios, they lack flexibility. For instance, systems like the *GliderAgent* that operate in constantly changing environment (e.g. cockpit of a glider), should not relay on compiled Java classes.

Next, we have considered rule-based expert systems found at [13]. While we report only those written in Java, *neither* of them satisfied our requirements. The primary concern was related to need to re-implement the RETE (e.g. in Roolie), or porting the RES to mobile devices (e.g. Drools, OpenL Tablets).

Finally, we considered two semantic frameworks – Apache Jena and the OWL API (in Section 2.3). Here, we decided to use Jena. First, it already did run on the Android OS. Moreover, the Jena on Android creators summarized the porting process, helping us to bring the newest version of Jena (2.11.1) to the Android OS. Moreover, because Jena implements the RETE algorithm, we can take advantages of two different frameworks – RES and semantic data processing. Specifically, the system knowledge can be represented and stored in the form of an RDF / OWL ontology, while the decision making process can utilize rule-based processing.

Porting Jena to Android proceeded in two stages: (1) creating a fully functional prototype, and (2) rewriting the Java code to run on the Android OS. Note that since the prototype was created first, we were able to remove unused dependencies and, in this way, reduce the amount of work in the second stage.

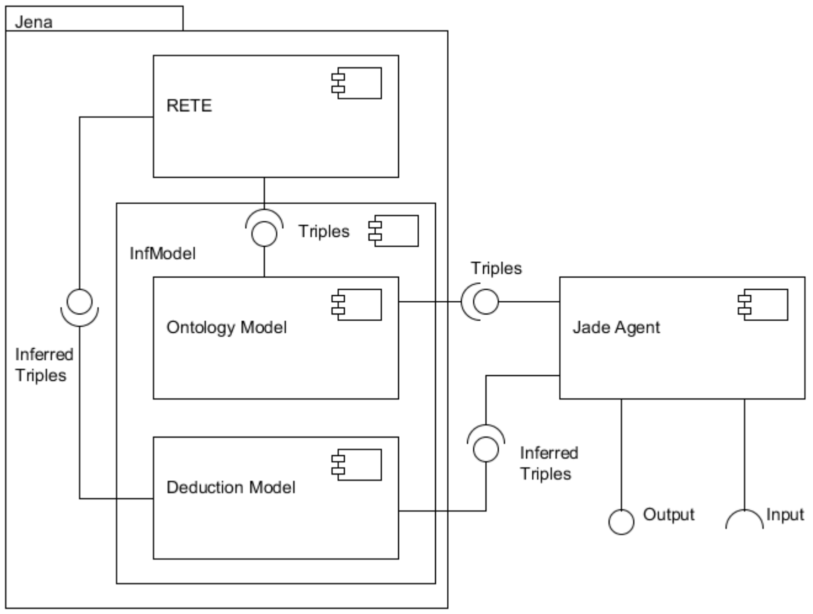


Fig. 1. The component diagram of the system

The component diagram is presented in Figure 1. There, we can see two software components: (1) Jena and (2) a Jade agent. During the initialization, a pair – an ontology and a set of rules – is loaded into Jena. They represent the knowledge base of the system. The ontology is stored as an instance of the *InfModel* class. Jade agent receives this information from the environment and analyzes it (extracts facts). Next, these facts are transformed into triples (object, predicate, subject) compliant with the loaded ontology. All triples are sent to Jena, in order to update the *InfModel*. Note that the *InfModel* is a hybrid of two different instances of the *Model* class: (i) the *Ontology Model*, and (ii) the *Deduction Model*. The *Ontology Model* stores all initial facts (loaded with the ontology) and the inserted triples (added by the Jade agent). On the other hand, the *Deduction Model* stores all facts inferred by matching rules against the ontology. When a new triple is added to the *Ontology Model*, Jena runs its implementation of the RETE algorithm. During this process, new inferred facts are added to the *Deduction Model*. Since a Jade agent is not "interested" in what it already knows (facts added to the *Ontology Model*), Jena returns new triples from the *Deduction Model*. In order to improve the system performance, these triples are combined into batches. A batch is sent when the algorithm completes the execution (all fulfilled rules were fired). Finally, the Jade agent analyses the batch and executes the appropriate behavior.

In the second stage, the system was rewritten to work with the Android API. First, it requires only a part of Jena functionality – for interacting with an ontology and executing the RETE algorithm. Thus, it was easy to empirically verify, that Jena requires the following libraries: i) jena-core, ii) jena-iri, iii) slf4j-api, iv) xercesImpl and v) xml-apis. These libraries can be roughly divided into three subsets: 1) Jena, 2) SLF4J, 3) Xerces. Here, SLF4J [19] and Xerces [20] are external projects. SLF4J (Simple Logging Facade for Java) serves as a simple abstraction for various logging frameworks. It allows the user to plug in the desired logging framework at the deployment time. According to the SLF4J website, there exists a wrapped implementation for the Android OS. Unfortunately, currently, the SLF4J is not available for download (the download site returns the 404 error). Since the Android API provides its own logger classes, it is not a crucial part of the application. Thus, during the implementation, we used the repacked SLF4J libraries from the Jade on Android project (which, however, may be outdated).

The Xerces (licensed to the Apache Software Foundation) is intended for creation and maintenance of XML parsers. It is a very important dependence in Jena and, thus, had to be rewritten. There exists Xerces for Android. This community-driven project is based on the latest version of Xerces (2.11.0) and is available for download at [21]. Unfortunately, Xerces for Android uses the *javax.** namespace to provide the missing dependencies. The *javax.** namespace is interpreted by the Dalvik (or ART) cross-compiler (as the "core" Java library), thus it is "safe" to cross-compile. To overcome this limitation, one can either compile the project with the "–core-library" flag (this suppresses the error in the compiler), or rename the *javax.** namespace to *javax2.** (as the developer of Jena on Android suggests). Overall, when SLF4J and Xerces are replaced, Jena can be repacked to be supported on the Android OS.

3.1 Testing the solution

To test our system we proposed two scenarios. In the first scenario, an agent system runs on a device with the latest stable version of the Android OS – 4.4.4. In the first step, initial facts and rules (presented in listing 1.1) are inserted into the system. These rules were written based on the theory on psychosocial development of human beings, articulated by Erik Erikson [6]. Then, we modify the system knowledge by interacting with the Android application. Specifically, we can increase or decrease the age by one. As a result, new triples (facts about our current age) are inserted to the system. In response to user actions, the agent display short information in the form of a "toast" notification (see Figure 2). After the last triple is delivered (the number is greater than 65), the system prints out the outcome in the debug console. Each part of the result contains the following information: the current age, the current Erikson's stage of human life and two triples (one inserted into the *Ontology Model* and one inserted into *Deduction Model*). Finally, we can observe that all stages were reached by the application. Specifically, the agent properly responded to all facts and the system knowledge was correctly updated by the algorithm.

Listing 1.1. Facts and rules used in the first scenario

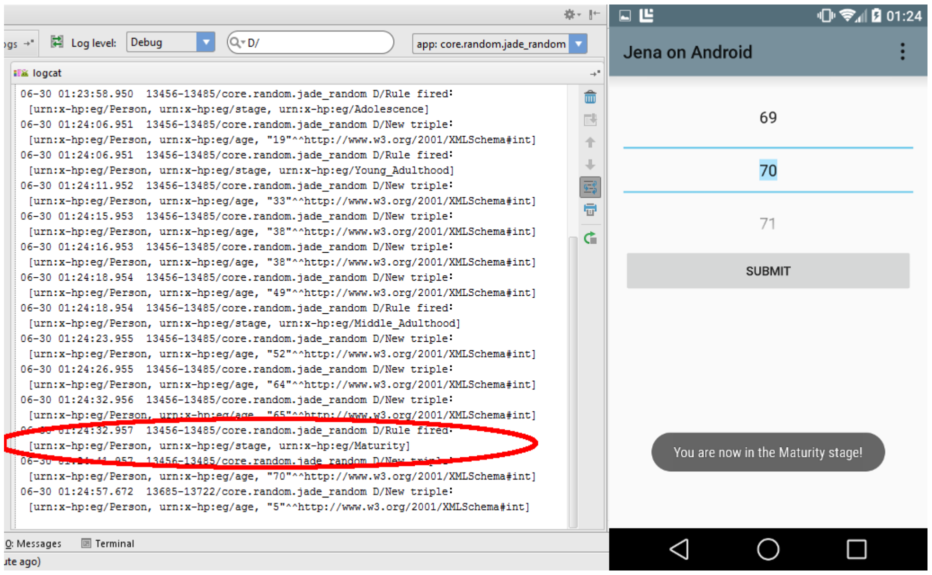
```
<!-- FACTS -->

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:eg="urn:x-hp:eg/" >
  <rdf:Description rdf:about="urn:x-hp:eg/Person">
    <eg:age rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</eg:age>
    <eg:stage rdf:resource="urn:x-hp:eg/Infancy"/>
  </rdf:Description>
</rdf:RDF>

<!-- RULES -->

[infancy: (?d eg:age ?a) ge(?a,0) lessThan(?a,2)
  -> (?d eg:stage eg:Infancy)]
[childhood: (?d eg:age ?a) ge(?a,2) lessThan(?a,3)
  -> (?d eg:stage eg:Childhood)]
[preschool: (?d eg:age ?a) ge(?a,3) lessThan(?a,6)
  -> (?d eg:stage eg:Preschool)]
[school: (?d eg:age ?a) ge(?a,6) lessThan(?a,12)
  -> (?d eg:stage eg:School)]
[adolescence : (?d eg:age ?a) ge(?a,12) lessThan(?a,19)
  -> (?d eg:stage eg:Adolescence)]
[young_adulthood: (?d eg:age ?a) ge(?a,19) lessThan(?a,40)
  -> (?d eg:stage eg:Young_Adulthood)]
[middle_adulthood: (?d eg:age ?a) ge(?a,40) lessThan(?a,65)
  -> (?d eg:stage eg:Middle_Adulthood)]
[maturity: (?d eg:age ?a) ge(?a,65)
  -> (?d eg:stage eg:Maturity)]
```

In the second scenario, the rule-based decision making system is integrated with the *GliderAgent* agent. Here, the system receives feeds from different sensors (altitude, temperature, blood pressure etc.) and, based on such information, triggers appropriate *GliderAgent* agent behaviors. Specifically, the rules and corresponding behaviors are listed in listing 1.2. Unlike the first scenario, the system takes autonomous actions. Specifically, for the test purposes, we used the oxygen scenario from the initial version of the system [23]. In this scenario, we model two types of warnings: (1) low oxygen level generated at 9842.52 ft (3000 m above the sea level), and (2) critical oxygen level generated at 13123.36 ft (4000 m above the sea level). At the beginning of the scenario, the glider stays on the ground at the altitude of 0 m. The position of the glider and its altitude start to change when the scenario is executed. It is assumed that the glider is conducting a lee-wave flight, and its altitude is increasing fast. Each time, when an agent receives data from sensors, the *Ontology Model* is modified accordingly. Namely, the position of the glider changes with respect to the GPS feed. Figure 3 presents the situation when the glider reaches the altitude of 4000 m. We can see that the



```
06-30 01:23:58.950  13456-13485/core.random.jade_random D/Rule fired:
 [urn:x-hp:eg/Person, urn:x-hp:eg/stage, urn:x-hp:eg/Adolescence]
06-30 01:24:06.951  13456-13485/core.random.jade_random D/New triple:
 [urn:x-hp:eg/Person, urn:x-hp:eg/age, "19"^^http://www.w3.org/2001/XMLSchema#int]
06-30 01:24:06.951  13456-13485/core.random.jade_random D/Rule fired:
 [urn:x-hp:eg/Person, urn:x-hp:eg/stage, urn:x-hp:eg/Young_Adulthood]
06-30 01:24:11.952  13456-13485/core.random.jade_random D/New triple:
 [urn:x-hp:eg/Person, urn:x-hp:eg/age, "33"^^http://www.w3.org/2001/XMLSchema#int]
06-30 01:24:15.953  13456-13485/core.random.jade_random D/New triple:
 [urn:x-hp:eg/Person, urn:x-hp:eg/age, "38"^^http://www.w3.org/2001/XMLSchema#int]
06-30 01:24:16.953  13456-13485/core.random.jade_random D/New triple:
 [urn:x-hp:eg/Person, urn:x-hp:eg/age, "38"^^http://www.w3.org/2001/XMLSchema#int]
06-30 01:24:18.954  13456-13485/core.random.jade_random D/New triple:
 [urn:x-hp:eg/Person, urn:x-hp:eg/age, "49"^^http://www.w3.org/2001/XMLSchema#int]
06-30 01:24:18.954  13456-13485/core.random.jade_random D/Rule fired:
 [urn:x-hp:eg/Person, urn:x-hp:eg/stage, urn:x-hp:eg/Middle_Adulthood]
06-30 01:24:23.955  13456-13485/core.random.jade_random D/New triple:
 [urn:x-hp:eg/Person, urn:x-hp:eg/age, "52"^^http://www.w3.org/2001/XMLSchema#int]
06-30 01:24:26.955  13456-13485/core.random.jade_random D/New triple:
 [urn:x-hp:eg/Person, urn:x-hp:eg/age, "64"^^http://www.w3.org/2001/XMLSchema#int]
06-30 01:24:32.956  13456-13485/core.random.jade_random D/New triple:
 [urn:x-hp:eg/Person, urn:x-hp:eg/age, "65"^^http://www.w3.org/2001/XMLSchema#int]
06-30 01:24:32.957  13456-13485/core.random.jade_random D/Rule fired:
 [urn:x-hp:eg/Person, urn:x-hp:eg/stage, urn:x-hp:eg/Maturity]
06-30 01:24:41.957  13456-13485/core.random.jade_random D/New triple:
 [urn:x-hp:eg/Person, urn:x-hp:eg/age, "70"^^http://www.w3.org/2001/XMLSchema#int]
06-30 01:24:57.672  13685-13722/core.random.jade_random D/New triple:
 [urn:x-hp:eg/Person, urn:x-hp:eg/age, "5"^^http://www.w3.org/2001/XMLSchema#int]
```

```
You are now in the Maturity stage!
```

Fig. 2. The outcome of the first scenario – the application properly notify the user

XCSoar program (which observes the altitude greater than 13123.36 ft) displays warning "Critical oxygen level". Overall, we can observe that the agent properly identified a life-treating situation and informed the pilot about the danger. Thus it can be said that the new *GliderAgent* agent "has its brain in place." This will also allow us to start building its knowledge base in form of rules and ontologies.

Listing 1.2. Facts and rules used in the second scenario

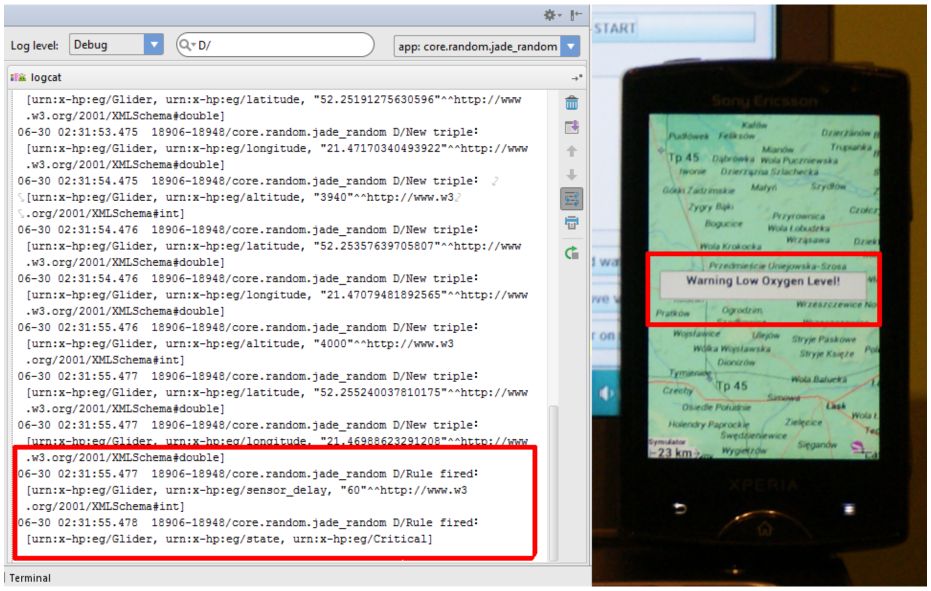
```
<!-- FACTS -->

<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:eg="urn:x-hp:eg/" >
  <rdf:Description rdf:about="urn:x-hp:eg/Glider">
    <eg:altitude rdf:datatype="http://www.w3.org/2001/XMLSchema#double">0</eg
        :altitude>
    <eg:latitude rdf:datatype="http://www.w3.org/2001/XMLSchema#double">0</eg
        :latitude>
  <eg:altitude rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</eg:
      altitude>
  (...)
  <eg:sensor_delay rdf:datatype="http://www.w3.org/2001/XMLSchema#int">300</
      eg:sensor_delay>
  <eg:state rdf:resource="urn:x-hp:eg/Normal"/>
  (...)
  </rdf:Description>
</rdf:RDF>

<!-- RULES -->

(...)
```

```
[low_oxygen: (?g eg:altitude ?a) ge(?a,3000) lessThan(?a,4000)
  -> (?g eg:sensor_delay 120) (?g eg:state eg:Cautious)]
[critical_oxygen: (?g eg:altitude ?a) ge(?a,4000)
  -> (?g eg:sensor_delay 60) (?g eg:state eg:Critical)]
(...)
```



```
 [urn:x-hp:eg/Glider, urn:x-hp:eg/latitude, "52.25191275630596"^^http://www
 .w3.org/2001/XMLSchema#double]
06-30 02:31:53.475  18906-18948/core.random.jade_random D/New triple:
 [urn:x-hp:eg/Glider, urn:x-hp:eg/longitude, "21.47170340493922"^^http://www
 .w3.org/2001/XMLSchema#double]
06-30 02:31:54.475  18906-18948/core.random.jade_random D/New triple:
 [urn:x-hp:eg/Glider, urn:x-hp:eg/altitude, "3940"^^http://www.w3
 .org/2001/XMLSchema#int]
06-30 02:31:54.476  18906-18948/core.random.jade_random D/New triple:
 [urn:x-hp:eg/Glider, urn:x-hp:eg/latitude, "52.25357639705807"^^http://www
 .w3.org/2001/XMLSchema#double]
06-30 02:31:54.476  18906-18948/core.random.jade_random D/New triple:
 [urn:x-hp:eg/Glider, urn:x-hp:eg/longitude, "21.47079481892565"^^http://www
 .w3.org/2001/XMLSchema#double]
06-30 02:31:55.476  18906-18948/core.random.jade_random D/New triple:
 [urn:x-hp:eg/Glider, urn:x-hp:eg/altitude, "4000"^^http://www.w3
 .org/2001/XMLSchema#int]
06-30 02:31:55.477  18906-18948/core.random.jade_random D/New triple:
 [urn:x-hp:eg/Glider, urn:x-hp:eg/latitude, "52.255240037810175"^^http://www
 .w3.org/2001/XMLSchema#double]
06-30 02:31:55.477  18906-18948/core.random.jade_random D/New triple:
 [urn:x-hp:eg/Glider, urn:x-hp:eg/longitude, "21.4698862329l208"^^http://www
 .w3.org/2001/XMLSchema#double]
06-30 02:31:55.477  18906-18948/core.random.jade_random D/Rule fired:
 [urn:x-hp:eg/Glider, urn:x-hp:eg/sensor_delay, "60"^^http://www.w3
 .org/2001/XMLSchema#int]
06-30 02:31:55.478  18906-18948/core.random.jade_random D/Rule fired:
 [urn:x-hp:eg/Glider, urn:x-hp:eg/state, urn:x-hp:eg/Critical]
```

Fig. 3. The outcome of the second scenario – the GliderAgent system is running on the Android device

4 Concluding remarks

In this paper, we considered implementation of "intelligent" software agents. Based on a analysis of possible approaches (native methods in agent platforms, rule-based expert systems and semantic frameworks), we have realized that none of them is sufficient alone, when developing agent systems for mobile devices and when agent knowledge has to be often updated. Therefore, we have combined Jade and Jena to develop a solution, which supports both rule-based and semantic technologies and tested the proposed approach on two simple scenarios. While the implemented solution is restricted to Java-based agents running on the Android OS, we believe that the presented results naturally generalize to other programming languages and operating systems.

References

1. Android os. http://www.android.com/.

2. Androjena. `https://code.google.com/p/androjena/`.
3. Apache jena. `http://openl-tablets.sourceforge.net/`.
4. Apache jena on android. `http://elite.polito.it/jena-on-android/`.
5. Drools. `http://drools.jboss.org/`.
6. Erikson's psychosocial stages summary chart. `http://psychology.about.com/library/bl_psychosocial_summary.htm`.
7. Fact++. `http://aosgrp.com/products/jack/`.
8. Hermit. `http://hermit-reasoner.com/`.
9. Jack. `http://aosgrp.com/products/jack/`.
10. Jadex. `http://sourceforge.net/projects/jadex/`.
11. Jason. `http://jade.tilab.com/`.
12. Java agent development framework. `http://jade.tilab.com/`.
13. Open source rule engines in java. `http://java-source.net/open-source/rule-engines`.
14. Openl tables. `http://openl-tablets.sourceforge.net/`.
15. Owl api. `http://owlapi.sourceforge.net/`.
16. Pellet. `http://clarkparsia.com/pellet/`.
17. Protege. `http://protege.stanford.edu/`.
18. Roolie. `http://roolie.sourceforge.net/`.
19. Simple logging facade for java. `http://www.slf4j.org/`.
20. Xerces. `http://xerces.apache.org/`.
21. Xerces for android. `https://code.google.com/p/xerces-for-android/`.
22. M.L. Brodie Dieter Fensel. *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce.* Springer, 2003.
23. Jan J. Domanski, Radoslaw Dziadkiewicz, Maria Ganzha, Andrzej Gab, Mariusz M. Mesjasz, and Marcin Paprzycki. Implementing glideragent—an agent-based decision support system for glider pilots. In *Software Agents, Agent Systems and Their Applications*, pages 222–244. 2012.
24. Charles Forgy. *On the efficient implementation of production systems.* PhD thesis, Thesis, Carnegie-Mellon University, 1979.
25. G. Frackowiak, M. Ganzha, M. Paprzycki, M. Szymczak, Y.-S. Han, and M.-W. Park. Adaptability in an agent-based virtual organization towards implementation. In Jos Cordeiro, Slimane Hammoudi, and Joaquim Filipe, editors, *Web Information Systems and Technologies*, volume 18 of *Lecture Notes in Business Information Processing*, pages 27–39. Springer Berlin Heidelberg, 2009.
26. Maria Ganzha and Jain C. Lakhmi. *Multiagent Systems and Applicatins.* A John Wiley and Sons, Ltd, 2009.
27. Mariusz Mesjasz, Domenico Cimadoro, Stefano Galzarano, Maria Ganzha, Giancarlo Fortino, and Marcin Paprzycki. Integrating jade and maps for the development of agent-based wsn applications. In Giancarlo Fortino, Costin Badica, Michele Malgeri, and Rainer Unland, editors, *Intelligent Distributed Computing VI*, volume 446 of *Studies in Computational Intelligence*, pages 211–220. Springer Berlin Heidelberg, 2013.
28. Hyacinth S. Nwana and Divine T. Ndumu. A perspective on software agents research. *Knowl. Eng. Rev.*, 14(2):125–142, 1999.
29. Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 2 edition, 2003.
30. John F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations.* Brooks / Cole, 1999.
31. Michael Wooldridge and Nicholas R Jennings. Intelligent agents: Theory and practice. *Knowledge engineering review*, 10(2):115–152, 1995.