Modeling cyber-physical systems – a GliderAgent *3.0 perspective*

Mariusz Marek Mesjasz, Maria Ganzha & Marcin Paprzycki

Journal of Intelligent Information Systems
Integrating Artificial Intelligence and Database Technologies

ISSN 0925-9902

J Intell Inf Syst
DOI 10.1007/s10844-019-00588-3

Springer

Journal of Intelligent Information Systems
https://doi.org/10.1007/s10844-019-00588-3

Modeling cyber-physical systems – a *GliderAgent* 3.0 perspective

Mariusz Marek Mesjasz[1] · **Maria Ganzha**[1] · **Marcin Paprzycki**[1]

Received: 12 October 2018 / Revised: 26 September 2019 / Accepted: 11 November 2019 /
Published online: 21 January 2020

Abstract
Eight years ago we have designed and implemented an initial prototype of an agent-based glider pilot support system (the *GliderAgent*). Our aim was to validate correctness of the initial assumption that an agent-based system, combined with sensor data, can help pilots in various situations that occur during a flight. For instance, the *GliderAgent* was capable of detecting certain dangers, warn the pilot and, autonomously, send notification(s) to the ground station. Due to the continuous and rapid development of mobile technologies and sensors, our initial prototype has evolved. First, we moved the system, from an emulated environment, to real devices. Second, a semantic-rule-based decision making system was integrated with the *GliderAgent*, to analyze feeds from sensors (altitude, temperature, blood pressure, etc.) and, based on received information, to trigger appropriate behaviors. The result of our work provided a foundations for development of a general-purpose, cyber-physical system framework, which will be described in this paper. Furthermore, the developed system illustrates an interesting approach to integration of heterogeneous IoT devices (potentially also IoT platforms).

Keywords Software agents · Decision support system · JADE · Android · Anomaly detection · Cyber-physical systems

1 Introduction

In 2009 we have started the development of an agent-based decision support system for glider pilots. The reason for this work came from an analysis of a series of glider accidents that took place in Poland and Czech Republic at that approximate time (for more details,

✉ Mariusz Marek Mesjasz
mesjaszm@ibspan.waw.pl

Maria Ganzha
maria.ganzha@ibspan.waw.pl

Marcin Paprzycki
paprzyck@ibspan.waw.pl

[1] Systems Research Institute, Polish Academy of Sciences, Newelska 6, 01-447, Warsaw, Poland

see Gab et al. (2010) and Domanski et al. (2012)) and our direct access to knowledge concerning the "world of gliding". Mentioned accidents happened regardless of the fact that pilots had standard pilotage software (e.g. the XCSoar (2019)) installed. However, it was easy to notice that, either it would have been possible to avoid some accidents, or their effects could have been mitigated if additional support was provided to the pilots and/or ground stations. The envisioned support was *not* to "replace" the pilot (e.g. by use of an auto-pilot), or the existing pilotage software, but to create an additional infrastructure that would autonomously react to, among others, danger situations, without the need for the pilot to do so; e.g. by informing the ground station that the crash is very likely (here, the pilot could focus her/his actions on avoiding/minimizing effects of the crash). The need for autonomy of actions suggested use of software agents (Jennings and Wooldridge 2001). As a result, the preliminary design of the system was completed and presented in Gab et al. (2010). Next, the design was turned into the initial working prototype (see, Domanski et al. (2012), for a complete discussion). The implemented system consisted of (a) the sensor part – SunSpot sensors (Sun SPOT 2019) emulated through the Solarium platform and managed through the MAPS agents, (b) agent part – *GliderAgent* agents implemented in JADE (Java agent development framework 2019), and (c) XCSoar pilotage software (XCSoar 2019) running on a PC. It has to be stressed that the system was running on top of an "emulator platform" that was also implemented. Over the years, the *Gateway*, one of the core parts of the system that allowed communication between JADE and MAPS agents, and the *GliderAgent* system itself were updated and improved (see, Mesjasz et al. (2013)). However, no major changes were introduced to the system itself.

In the meantime, a number of changes took place in computer technologies. The most important of them (in the context of our work) were: proliferation of (i) smartphones, (ii) wearable devices (such as wristbands and watches), (iii) high-end single-board computers, and (iv) tracking platforms for small aircrafts. Smartphones and wearables have successfully replaced Personal Data Assistants (PDAs) and opened new possibilities by popularizing wearable sensors. Single-board computers (such as Raspberry Pi (2019), Banana Pi (2019), etc.) gradually supplanted other devices (for example, the outdated SunSpots) from the market. It became clear that the *GliderAgent* system has to evolve, to catch up with changes in "computer hardware" and "intelligent software." Furthermore, the JADE agent platform has progressed to include more realistic support for running agents on mobile devices (i.e. the JADEAndroid module). Several software updates, including a number of them in the XCSoar pilotage software, have been released. Finally, a number of on-line tracking platforms (i.e. SkyLines, Open Glider Network) emerged and provided significant advancements in on-line tracking hardware and software for small aircrafts. All these changes resulted in a need to redesign, partially re-implement and update the *GliderAgent* system.

Separately, a more general reflection concerning state of the "world of computing" made us realize that the *GliderAgent* system is actually an instance of a more general class of, so-called, Cyber-physical systems (2012). By analyzing the cyber-physicsl system concept map available at Cyber-physical systems (2012), one can observe that most of its nodes are currently covered by the *GliderAgent* system. Namely, it works in real-time, monitoring and analyzing sensor feed. The human element plays a key role – the interaction with the user and her/his feedback allows the system to understand the context. By the nature of an agent-driven software, the *GliderAgent* agent is intelligent, adaptive and proactive. Therefore, we have decided that, while updating the *GliderAgent* system, we will also "generalize it" in a way that will allow it to become easily adaptable to realize other cyber-physical systems/scenarios.

In turn, this activity allowed us to suggest an approach to dealing with interconnection of heterogeneous Internet of Things devices (and, possibly, integration of heterogeneous Internet of Things platforms). Agreeing with the conceptualization of relationship between the Internet of Things (IoT) and the Cyber-Physical Systems (CPS) found in IDEAS'2020: A Tour of Tomorrow's World (2019) (as well as in Internet of things towards ubiquitous and mobile computing (2019) slide 10), we see the IoT as the physical ("low-level") part of the CPS. Hence, for instance, heterogeneous wearable and battery-powered sensors found in the glider represent the IoT layer of our CPS. This being the case, the *GliderAgent* system, being capable of (re)acting on the basis of feeds from such sensors represents a method of making them interoperable. Furthermore, assuming that groups of sensors involved in gliding would be combined into IoT platforms (e.g. personal sensors, equipment-related sensors, airfield sensors, etc.) the proposed approach illustrates a potential way of combining them.

The aim of this paper is to provide detailed technical information about the results of the, most recent, *GliderAgent* system redesign and to present our solutions to challenges that we have faced during the development. With our latest iteration of the *GliderAgent* system, we adapted our initial prototype to the current state-of-the-art in many research areas – especially by replacing obsolete SunSpot sensors with modern IoT devices and introducing an ontology with the rule-based reasoning. These changes make this paper a case study of practical aspects of working with software agent technologies within a CPS/IoT system. In this way it is also a response to a lingering gap between theoretical work focused on advancing agent system theory, and the practice of use of software agents in the real world (see, Nwana and Ndumu (1999), Belecheanu et al. (2006), and Ganzha and Lakhmi (2009)). The main message originating from these sources is that, in agent literature, there is not enough discussions of *practical aspects* of development and implementation of agent systems. Furthermore, instead of focusing on ideas that are supported by small-scale prototypes, it is of great value to advancement of agent systems to develop complete system prototype, even if only with a limited functionality of its individual parts. These observations provided the methodological underpinning of our work, as we have developed and (re)implemented a prototype of a complete system, and we report on our findings.

To this effect we start from outlining key issues involved in glider flying. We follow with summary of the state-of-the-art in all pertinent areas. Next, in Section 4, we describe key facts concerning the development of the original *GliderAgent* system. This description is followed by presentation of technical aspects of changes that led to the development of *GatewayAgent* version 3.0. Brief description of tests performed with the current version of the system are presented in Section 6. Next, we discuss how the *GliderAgent* 3.0 became a general-purpose CPS platform and how it represents an attempt at solving a problem of interconnection heterogeneous IoT devices. Summary of material and suggestion of future research directions complete the paper.

2 Need for glider pilot support

Let us start from a very brief introduction to glider flying and concerns that led us to development of the original *GliderAgent* system (for more details, see Gab et al. (2010) and Domanski et al. (2012)). Before we proceed, let us note that, while the material presented here concerns glider pilots, it can be easily extended to other forms of ultralight/microlight aviation (Airspace in the U.S. 2019).

2.1 Electronic equipment in gliding

Computer technology provides only minimal support for glider pilots. On board of gliders, there are radios, which are the main and the oldest electronics on board. They provide the basic means of communication between gliders, and between gliders and the ground station(s).

Furthermore, we can often find so-called loggers, which collect data about the flight. They are particularly important in the case of sport// flying (see, Section 2.2), but they do not directly support the pilot in her/his actions.

Moreover, glider anti-collision systems are in the works. For example, the FLARM system (2019), deployed primarily in German speaking countries, Scandinavia, as well as in Australia, and New Zealand. Such systems are designed for a single functionality and have very narrow communication bandwidth (needed to provide collision warning).

Separately, we observe growing interest in "on-line tracking," which visualizes positions of gliders during competitions. Here, the goal is to make gliding (as a sport discipline) more audience-friendly. Note that tracking systems deliver only a single (tracking) service, and do not support the glider pilot. Furthermore, since only a relatively narrow communication bandwidth is needed for tracking, they cannot be naturally extended to support other functions.

2.2 Typology of glider flying

Let us now enumerate basic forms of glider flying, as they provide context for the decision making (see, also Gab et al. (2010) and Domanski et al. (2012)).

1. *Training flights*, which can be divided into:
   a) *Flights with an instructor*—early-training flights where a student gains basic skills—how to start, to land, etc.
   b) *Solo training flights*—conducted by a student alone, while an instructor is located on the ground; solo training flights usually take place within no more than 20 km from the airfield.
2. *Recreational flying*—involve long distances from the start airfield (more than 20km); very often conducted over areas not well-known to the pilot; three sub-categories can be distinguished:
   a) *Cross-country flying*—the easiest and safest form; typically flown over plains; awareness of other gliders is crucial to avoid collisions.
   b) *Ridge flying*—demands high skills because of flying close to slopes of hills and/or mountains; characterized by high variability of flying conditions (multiple lifts and sinks); terrain characteristics (lack of landing space) exaggerates possible dangers.
   c) *Lee-wave flying*—involves a dynamic phenomenon that allows gaining several kilometers of heights, and make flights hundreds of kilometers long; characterized by violent conditions for start, getting to the wave and landing; possibility of forming "a solid cloud," which makes returning to the ground very difficult, adds to encountered dangers.
3. *Sport/competitive flying*—conducted in accordance with specified rules and procedures (to be accepted as a sport). Among sport events we can distinguish:

a) *Sport badges and diplomas*—achieved, for instance, for completing soaring performances of: distance, duration and height.
b) *Localized competitions*—pilots fly together starting from the same place.
c) *On-line contests*—pilots conduct flights in their clubs and send documentation to the contest organizer.
d) *Record flights*—establishing national or world records; validated by an appropriate authority.

2.3 Supporting glider pilots – foundations

Combining above presented material, with analysis of incidents involving gliders, presented in Gab et al. (2010) and Domanski et al. (2012), let us identify some of the key support areas, arising in flying gliders. Obviously, this list is not exhaustive, for more information, see Gab et al. (2010) and Domanski et al. (2012).

- In *training flights* it is important to instruct the student pilot. Here, we can distinguish the advice made available during the flight, and post-flight analysis. During the flight, the instructor could make suggestions while observing parameters transferred from the glider. In a post-flight session, instructor and student could re-play and analyze data collected during the flight.
- In a *cross-country* flying, help finding thermals would be useful (e.g. based on information from other gliders). Furthermore, informing the ground (in advance) that the glider cannot reach any airfield could improve flight safety. Finally, pilots could use information like, for instance, meteo reports, especially warnings about possible storms.
- *Ridge flying* safety requires (fast) informing the ground on abnormal behavior(s) or danger(s). Not only the mountainous area is not appropriate for landing, but the time between appearance of the danger and the crash is much shorter than in cross-country flying.
- Flights using the *lee-wave* phenomenon involves, among others, reaching heights (above the 4000 meters) where lack of oxygen could result in altitude sickness (Altitude sickness 2019) and loss of consciousness (see Gab et al. (2010), for the description of an actual accident caused by oxygen deprivation). This further increases the need for pilot support, as the danger becomes more pronounced than in the case of *ridge flying*.
- Finally, in *sport// flying* all the above mentioned needs can materialize, as the competitions are conducted over the fields, and in the mountains. Furthermore, it is important to make competitive glider flying interesting to the spectators.

3 State-of-the-art in related areas

Let us now summarize the state of the art in the main "technological areas" that are related to the development of the *GliderAgent* system (its earlier versions, and the current one).

Software supporting glider pilots Nowadays, "glider software" falls into one of two categories: (i) simulators and (ii) glider computers. Simulators create a virtual reality, in which aviation enthusiasts can fly an aircraft (not necessarily a glider, but almost any plane existing in the world) to learn, test their skills, or have fun. Note that flight simulation itself is not our goal, and this is the key difference between such software and the *GliderAgent*.

Our system is meant to operate in an actual glider and support the glider pilot. Henceforth, a "simulation" is used only for the purpose of evaluating the correctness of the developed system.

Glider Computers, such as SOAR (2019), SeeYou Mobile (2019), WinPilot (2019), or XCSoar (2019) provide very similar functionality to the *GliderAgent*. However, there are significant differences:

SOAR is a Windows based flight planning and analysis program. It can be used for planning cross-country flights and analyzing GPS recordings. SOAR (as well as other glider planners), does not work on an actual glider. Thus, the pilot can only use it before or after a flight. This, obviously, is different from our system designed to monitor the environment (geoposition, collision possibility, etc.) in a glider.

SeeYou Mobile is an in-flight navigation program designed for PDA and PNA devices, with Windows Mobile or Windows CE operating systems. It generates visual and audio warnings about the airspace in the vicinity of the glider. However, this function is strictly limited to "forbidden areas" (hard-coded into the device). In comparison to the *GliderAgent*'s collision detection, SeeMobile does not track nearby air-traffic.

WinPilot and XCsoar resemble the car GPS navigation, designed for pilots. The WinPilot can run on Windows Mobile, Pocket PC and iOS operating systems. It is a closed-code software (requiring license) maintained by the Sierra SkyWare. The original XCSoar was a commercial software developed for the Pocket PC. In 2005, the program became an open-source, community-driven project published under the GNU General Public License. Nowadays, it is actively developed by a group of volunteers and community members (the last release took place in August 2018). Moreover, developers can modify its publicly available source code to provide functionality beyond its original scope. Currently, the XCSoar runs on Windows, Windows Mobile, Unix, and Android devices. These features make it the most successful and popular tactical glide computer. XCSoar is also a client for SkyLines (2019). SkyLines is an internet platform that allows flight sharing. Unfortunately, SkyLines data is only available online via the official website.

Open glider network The*Open Glider Network* (*OGN*) is a community project, which aims at creating a unified tracking platform for small aircrafts (i.e. gliders, helicopters, drones). The platform consists of software and hardware that are used to communicate with *OGN* servers. Each small aircraft (in our case, a glider) is to be equipped with an **OGN** tracker. **OGN** receivers, placed on the ground, collect data sent by *OGN* beacons and forward them to *OGN* servers. The collected data can be accessed by an API.

Thanks to the active *OGN* community, new receivers are installed in gliding clubs, airfields and mountain areas. Each *OGN* antenna (the detailed design provided on the *OGN* website) can receive a signal form an aircraft at a distance of 130+ km. The coverage is especially good in western Europe (Germany, France, or United Kingdom) and in the eastern part of Australia.

The *OGN* infrastructure is very similar to the *GliderAgent* environment, proposed in Gab et al. (2010). Namely, *OGN* receivers collect data about gliders (including their GPS location) and send them to the *OGN* server. There data can be accessed and analysed by, for example, a Search And Rescue (*SAR*) service. Each device is registered in the *OGN* network, and has a unique ID. These IDs can by used for filtering gliders during competitions. However, since all the communication is unidirectional (from a beacon through a receiver to a *OGN* server), the *OGN* platform does not allow gliders to communicate with each other. The *OGN* tracking protocol (http://wiki.glidernet.org/ogn-tracking-protocol) sends glider

meta data (i.e. an aircraft type) and current flight parameters. However, lack of pilot's biometric data stands in contrast to the proposed *GliderAgent* system that also takes pilot's current state as an important factor.

Since *OGN* already has a number of receivers all over the world, extending the well-established *OGN* protocol and keeping it compliant with the existing infrastructure may prove to be difficult (additional bytes can be seen as "damaged" radio frames by receivers, or be dropped by servers). Thus, creating an incompatible radio frame may result in losing the benefit of wide radio-coverage.

"Intelligence" in flying/gliding In scientific literature, one can find several references to "intelligent soaring." However, most of them involve autonomous flights; e.g. drones/UAV's flying without interaction with the user. For example, in Atkins et al. (1999), authors focus on achieving fully automated aircraft flights. Despite the fact that the proposed system faces many similar challenges (for instance, "real time" monitoring, collision detection), our work represents a user-centric approach. The *GliderAgent* system aids the glider pilot during flight, but always leaves the final decision to her. On the other hand, other gliding support systems have rather limited functionality. For example, Hook et al. (2007) is dedicated to determining when and where to seek a lift. In case of the *GliderAgent*, collecting data for further analysis is only *one* of its possible functions.

Mobile devices Today, mobile devices play more and more important role in daily life. As smart-phones and tables become more powerful, they gradually replace personal computers for everyday use. Currently, there main mobile OSs are: Android (74.85%), iOS (22.94%) and others (2.21%) (IDC: Analyze the Future 2019).

The most popular of them (and thus our choice), Android (2019), is an open source, Linux-based, operating system developed by Google (2019) for touch-screen mobile devices such as smartphones and tablets. The source code of the Android is provided by Google under the Apache License (2019). Today, Android is expanding to cars, TVs and wearable devices. The latest release, codename "Oreo", appeared in August 2017 (Android lollipop 2019).

Personal sensors In the last few years, there has been a significant increase in the number of personal sensors. Nowadays, people often wear various smart-bands, smart-watches and other wearable devices (Apple watch 2019; Android wearable 2019). For example, smart-bands with a heart-beat monitor (sensor) collect information about their user' heart rate. Modern wearable devices, can easily share such information. For example, a smart-band, which is connected to a smart-phone via Bluetooth, can relay collected information on the server for further analysis. This phenomenon resulted in many publications concerning Body Sensor Network (BSN) (Fortino et al. 2013), that can collect and process huge amount of data from different body sensors.

In the *GliderAgent* system, pilot's health is an important factor. Each pilot's body can behave differently in the same conditions. Some accidents are results of pilot's fainting out in the cockpit (Gab et al. 2010). This is especially true for competitive flying where the pilot is encouraged to push his/her limits. The advancements of and accessibility to body sensors improves *GliderAgent*'s capabilities to monitor and warn the pilot before such life-threatening situation occur.

New generation of computer hardware A new class of devices that resulted from current trends in technology (miniaturization, development of smart-phones hardware caused by

increasing number of mobile devices) are single-board computers (*SBC*). The *SBC*s are built on a single circuit board. On such board, microprocessor(s), memory, input/output (I/O) and other features required of a "complete computer" are placed. There are several popular choices among SBCs (depending on hardware specification). Examples of cheap, yet high-end SBCs, are Raspberry Pi (2019) and Arduino (2019) (known as Genuino outside US). Both computers are actively developed – they were released in June 2019 (Raspberry Pi 4 Model B) and March 2019 (Arduino IDE version 1.8.9) respectively.

Software agent platforms Finally, let us discuss the pertinent state of the art in software agents. Recall, that the *GliderAgent* is based on agent technology.

(1) *Mobile-C* (2019) is an Foundation for Intelligent Physical Agents (2019) standards-compliant multi-agent platform written entirely in C/C++. This platform was specifically designed for real-time, resource constrained applications, with an interface to the hardware. As a result, the Mobile-C is focused on development of small/lightweight agents, which would fit the sensor part of our system, but not match well the needed decision support functionality. The latest update to v2.1.5 of was on 12th August 2014, which materialized after a period of about 3 years of dormancy. This makes it a bit suspicious as far as future of this platform. From March 2017, *Mobile-C* is also available on Raspberry Pi and other ARM-based computers. The platform would make a good candidate for our "sensor" part of the *GliderAgent*. Unfortunately, our code-base is written in JADE and replacing the existing JAVA-based components would be a huge undertaking which the current development cycle of *Mobile-C* does not justify. Namely, *Mobile-C* focuses on low-cost modular robots with almost no feature-rich updated to the core platform in the 5 years span.

(2) *JADEX* (2019) (latest stable update to v3.0.115 was released in January 2019) is an agent platform that was developed on the basis of the Belief Desire Intention (BDI) approach. It allows implementation of intelligent software agents using XML and Java. While JADEX meets our requirements concerning decision support and capability of running on Android, it has two major disadvantages: 1) it is systematically switching its focus to agent-as-component programming, which may-or-may-not provide the right level of granularity for our systems, and 2) writing the BDI rules in XML is quite a cumbersome and error prone process.

(2) *JASON* (2019) (latest stable update to v2.4 was released in May 2019) is interpreter for an extended version of AgentSpeak. AgentSpeak is a Belief Desire Intention (BDI) agent-oriented logic programming language. While *JASON* is a powerful tool, it is still an extension that has to be utilized within a multi-agent system like *JADE*. Moreover, there are no evidence of *JASON* being capable to run on a mobile device.

(3) *JADE* (2019), is a FIPA-compliant open-source framework for writing agent systems in JAVA. The JADE has a community of developers who create add-ons for it. Currently, JADE runs on Windows, Unix and Android devices. Moreover, based on the experimental research present in Chmiel et al. (2005), JADE is a highly scalable agent platform limited only by the standard limitation of the JAVA programming language. The latest update to v4.5 was released on 8th June 2017 (since its authors consider JADE to be "feature complete", in the future, only maintenance releases are planned). Moreover, JADE agents can be compared to the Android OS (Pintea et al. 2018) (recall that running the *GliderAgent* system on mobile devices is one of our initial goals). From the comparison, an observation can be made that there are many common features between both platforms. Namely, both platforms utilize independent

components that asynchronously communicate and cooperate with each other by sending unified messages. These similarities in the architecture and functions can provide to a interesting synergies and enhance a possible JADE-Android applications.

4 Architecture of the GliderAgent system

4.1 Summary of functionality

From our analysis presented in the Section 2, the *GliderAgent* system has been designed to: (a) help the pilot in navigation and pilotage, (b) deliver additional flight information, (c) provide support in emergency situations, and (d) facilitate communication, monitoring, logging events, etc. Let us stress, again that it was *not* designed to replace the existing systems (e.g. the XCSoar), but to *integrate* with them and *extend* their capabilities. Let us outline the most important requirements underlining our design. The following material should be matched with the use case diagram presented in Fig. 1 (and Domanski et al. (2012) Fig. 1).

– *Sharing information*. The system should allow for messages to be passed between various actors. Within a glider cockpit, a main unit (probably a smart-phone) should communicate with sensors placed within the glider and on the pilot. The data collected from sensor is to be aggregated and used in the decision making process. Beyond the cockpit, the glider should communicate with other gliders and ground stations. In places with no natural radio-coverage (i.e. mountains), the system should utilize, so called,

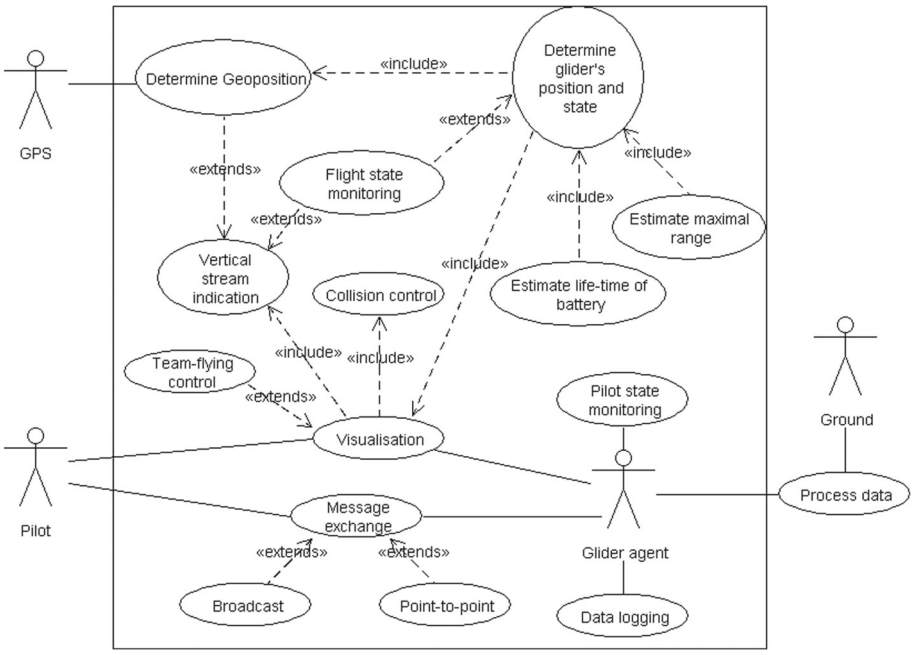

Fig. 1 Use case diagram of the GliderAgent system

"proxy-stations" which extends its reach. By sharing these message, the *GliderAgent* system should 1) avoid mid-air collisions, 2) alarm Search and Rescue (*SAR*) services in case of life-treating situations and 3) log flights for further analysis (finding points of interest, monitoring training flights etc.). The glider should also have access to more advanced data (often unreachable without the Internet) like the weather forecasts and warnings.

- *Pilot state monitoring*. The system should assess the "state" of the pilot. For this purpose, it should track selected biological parameters (using appropriate bio-sensors), and compare them to the norms. In selected contexts (e.g. perceived danger), results of monitoring should be used to warn the pilot and, autonomously, communicated to the ground station.
- *Estimated life-time of the battery*. The on-board electrical systems can function only if they have energy. The system should estimate how much energy is left in the battery, and warn the pilot (and possibly the ground station) when the battery is about to run out. The energy-level, and the type of flight, provide the context for energy saving decisions (e.g. when to pass a message to another glider, and when to preserve energy).
- *Flight state monitoring*. The system should track gliders (keep track of their flight states). This includes, among others, position, speed vector, acceleration. This information should help avoid crashes, observe student flights, gather data about a flight for further use (see, data logging, below), etc. Here, the idea is to directly extend capabilities of existing glider-GPS-software.
- *Vertical stream indication*. While software, like the XCsoar, indicates locations of vertical streams, the system should use glider-glider communication to share this information.
- *Collision detection*. The collision detection functionality should be based on exchanging location information between gliders.
- *Maximal range*. The system should help the user to estimate the maximal range that can be reached by the glider in given conditions. It would help properly plan the flight, and react to specific conditions (e.g. inability to reach the nearest airfield; see, the accident description presented in Domanski et al. (2012)).
- *Data logging*. Flight and pilot data should be logged and gathered for future analysis. Since the glider may have limited data-space available, depending on the scenario, selected information should be forwarded to be stored at the ground station.

4.2 GliderAgent 1.0 – implementation and testing

The system depicted in Fig. 1 has been turned into a prototype. During the implementation process, the following software was used:

- The main part of the system was implemented using the Java Agent Development framework (JADE) version 4.0.1.
- The Mobile Agent Platform for Sun SPOT devices (MAPS; Aiello et al. (2011)) was selected for management of sensor data. Here, the Sun SPOT connected sensors were envisioned as sources of information about the state of the pilot, as well as other data, e.g. available battery power.
- A JADE-MAPS gateway was implemented to address the fact that both agent platforms use different agent management (naming) and agent communication mechanisms. JADE uses FIPA AMS for agent management and ACL messages for communication.

MAPS uses Execution Engine for agent management and a simple set of properties encapsulated within a structure called MAPS Event for communication (for more details, see Domanski et al. (2012) and Mesjasz et al. (2013) and Section 5.1).

– SunSpot SDK (Sun SPOT 2019; Sun Small Programmable Object Technology (Sun SPOT) 2012), with an emulator – Solarium 6.0 – was used to develop the WSN subsystem.
– The XCSoar (2019; version 6.0) was extended. Since the XCSoar is written in C++, while the JADE platform is Java based, a bridge between the two was implemented to allow them to exchange data.
– Swingx-ws (Web Oriented JavaBeans and Swing (2019), version 1.0) was used to display a map including locations of gliders as seen by the ground station.
– Finally, the OpenStreetMaps (2019) software was used for mapping. The Swingx-ws's JXMapKit dynamically loaded data from the OpenStreetMaps server. Therefore, this function required an active Internet connection.

Note that some key system implementation decisions were forced by the state of available computer technology. While it was assumed that the agents supporting the pilot will run on her/his smartphone/PDA, at this time very few smartphones were capable of actually running (JADE) agents (without an enormous effort; for instance, trying to make it happen using the LEAP (Caire and Pieri 2011)). Therefore, it was decided that the *GliderAgent* 1.0 will work entirely in an emulated environment (GliderAgent Simulation) running on a personal computer (desktop or laptop). This environment not only provided the "external" data needed for the *GliderAgent* to be experimented with (e.g. the glider position and altitude, or the battery energy level), but also emulated the smartphone/PDA device itself.

The component diagram of the original simulation environment is presented in Fig. 2. The GliderAgent Simulation is composed of three main components: 1) GliderAgent, 2) GPS Server, and 3) Sensor Data Server. The GliderAgent agent is a JADE software agent, which analyses incoming data and, based on their content, supports the glider pilot. In other words, it can be seen as a pilot's personal agent that runs on her/his smartphone.

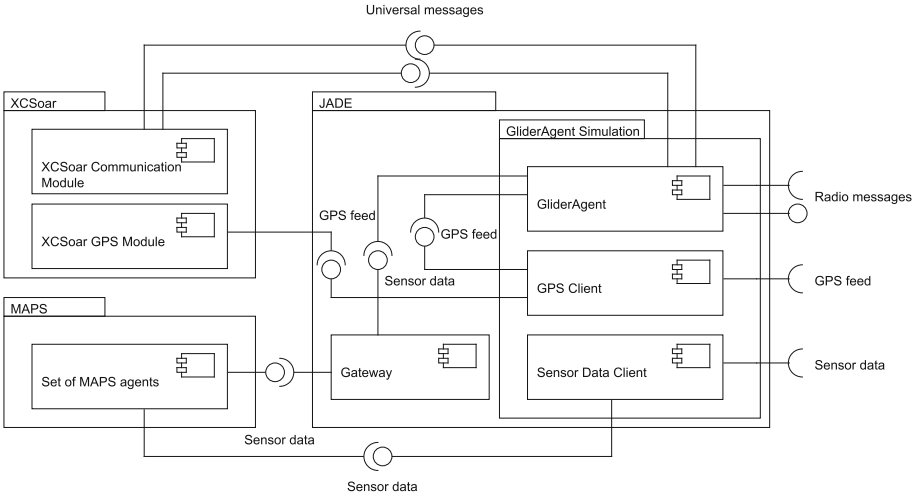

Fig. 2 Component diagram of the GliderAgent 1.0 simulation environment

The incoming data includes emulated radio messages from other entities in the system (e.g. Groundstation(s), and other GliderAgent agents), the scenario GPS feed (provided by the GPS Server; emulating glider's movement in space), and sensor data provided by the Sensor Data Server. The feed from the GPS Server is sent concurrently to the GliderAgent and to the XCSoar. The last element of the GliderAgent Simulation is the Sensor Data Server. The Sensor Data Server is not directly connected to the GliderAgent, but it feeds the data to the Solarium-emulated sensors (for example, pilot's oxygen level or blood pressure, or battery state). This data is then forwarded to the MAPS agents. Next, the MAPS agents forward the selected information to the GliderAgent, via the JADE-MAPS Gateway. Observe that the developed system has the general structure of a CPS. First, the GPS Server and the Sensor Data Server can be seen as two IoT platforms, while the GliderAgent agent represents the Cyber part of the CPS (see, also Internet of things towards ubiquitous and mobile computing (2019); slide 10).

Note that the *GliderAgent* system does not interact with the pilot directly. These interactions are achieved through the XCSoar software, which has been extended to interface with the *GliderAgent*. In other words, content delivered by GliderAgent is displayed "within" the XCSoar display. Moreover, to perform advanced calculations concerning the flight, the *GliderAgent* utilizes the XCSoar engine.

The implemented prototype has been tested in multiple use case scenarios. Description of these scenarios, as well as results of tests, can be found in Domanski et al. (2012). Regardless of the limitations imposed by the technology, the initial version of the system worked according to the expectations. Let us now proceed with the description of changes and modifications made in the system.

5 Key novel elements of *GliderAgent* 2.0

Let us now describe, in some detail, new elements that have been updated/implemented in the *GliderAgent* release 2.0. First, we have updated the JADE-MAPS gateway. Results of tests performed to evaluate its efficiency have shown weaknesses of the initial implementation. The most important problem was the delay between receiving an ACL message from the JADE subsystem and sending corresponding MAPS Events to the specified MAPS agent(s). In addition, we have generalized the gateway, allowing it to be used also when other software interacts with JADE agents. We discuss these changes in Section 5.1.

Second, we made the *GliderAgent* run on actual mobile devices. This functionality became available thanks to two developments. First, proliferation (and steady increase of capabilities/power) of Android-based smartphones (see, Section 3). Separately, let us recall that the concept of a Personal Data Assistant (PDA), understood as an independent device running calendaring applications, featuring news and games, has been subsumed by the smartphones. Second, implementation of the JADEAndroid add-on for JADE (see, JADE Android add-on Guide (2019)). The key feature introduced in the JADEAndroid was the ability to run the full JADE agent platform in a stand-alone mode. Here, while trying to port the *GliderAgent* into an Android smartphone, we have found that using JADEAndroid when JADE is run in a stand-alone container requires extra work, reported in Section 5.2.

Third, with the increasing number of on-line tracking systems, and the general advancement in hardware for small aircrafts (like the one provided by the Open Glider Network project), the decision was made to focus on a "intelligent" aspect of the *GliderAgent* system. Since none of the analysed platforms (see Section 3) focused on aiding an individual

glider pilot during a flight, our research in this area was a novelty with potential to benefit the entire gliding community and the research area of software agents in general, more than becoming a competitor to other fast-growing hardware-driven initiatives concerning the radio communication and data transmission. Hence, we have analyzed ways of introducing "brains" into software agents. Specifically, we have considered placing intelligence as an "agent code," using rule based expert systems, and semantic technologies. The results of this analysis have been presented in Mesjasz et al. (2013). In Section 5.3, we summarize our findings and show that our decision holds also today.

Upon completion of the *GliderAgent* system update, we have successfully run the same set of tests as we did for the original prototype. Since our ultimate goal, is to discuss the most current version of the *GliderAgent* system, and to show how it became a general cyber-physical platform (interconnecting heterogeneous IoT devices), in the next three section we will discuss the *current* (used in *GliderAgent* 3.0) state of: (i) the Gateway, (ii) running JADE agents on Android, and (iii) inserting decision-making capabilities into JADE agents.

5.1 A general purpose gateway

As stated above, the *JADE-MAPS gateway* (or, within this Section, the *Gateway*) has been implemented to provide a communication mechanism between JADE and MAPS agents. It facilitated bi-directional translation between *JADE ACL messages* and *MAPS Events*, and supported routing of communication between the two agent platforms. Let us now discuss, in more detail, the new design of the gateway that responded to (i) results of experiments performed with GliderAgent 1.0 (and 2.0), and (ii) the need of replacing MAPS agents with another "source of sensor information." The experiments reported in Mesjasz et al. (2013) pointed to two problems:

- If the number of simultaneously communicating agents exceeds a certain level, a communication bottleneck has been observed.
- The JADE-MAPS interaction could have been simplified by using novel mechanisms provided in the latest version of the JADE agent platform.

The first problem was addressed by significantly simplifying the work-flow of the *Gateway*. Instead of using time-consuming methods from the MAPS platform library, a new light-weight communication module was introduced. The module utilized methods from the SunSpot library (direct control over the radio channel) and performed only a minimal number of operations that ensured the data frame was acceptable and understood by the MAPS deceives. As a result, time between receiving and sending the message was greatly reduced. The second problem was partially addressed by re-factoring the existing *Gateway* code. The *Gateway* was decomposed into a general purpose *Gateway agent* and a separate MAPS-specific engine. Since MAPS logic was encapsulated in an interchangeable module, this was also the first step towards generalization of the *Gateway*. Namely, starting from that moment, the *Gateway* could connect the JADE platform with "everything" that has a suitable communication module (e.g. a device, an agent platform, etc.).

Addressing these two problems turned out to be a pivot point in prolonging the *GliderAgent* system lifespan. When in late 2014, Oracle has officially stopped selling and supporting Sun SPOTs, a new replacement module was implemented for the *Gateway agent*. SunSpots were replaced by the Banana Pi (2019). Since Banana Pi has sufficiently higher hardware specification than the SunSpot device, and allows to connect sensors via general purpose input/output (*GPIO*), it was a natural choice. Note that the Banana Pi was chosen over the Raspberry Pi due to the amount of available RAM, and dual core CPU, within the

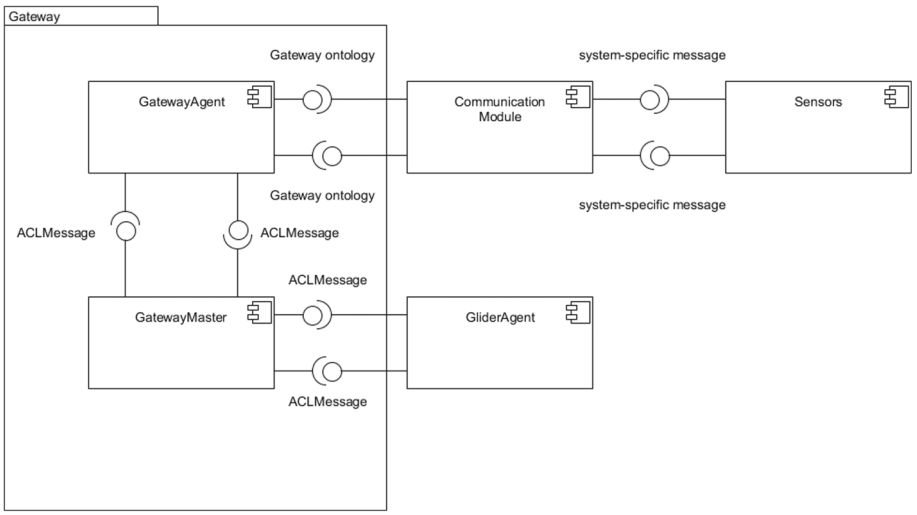

Fig. 3 A component diagram of the gateway

same price tag. However, let us stress that the following discussion applies to any other SBC that can run Linux-based system with Java SE for the ARM architecture (see, also Java se for arm (2019)). Arduino was rejected, because it comes with preinstalled software and low-spec hardware. When having IDE from hardware creators is beneficial for some projects, in our case, Arduino would make us rewrite entire system (even parts that do not require any update) (Fig. 3).

The component diagram, presented in Fig. 4, depicts the *Gateway*, used in the *GliderAgent* system 3.0. Here, the *Gateway* consists of a set of JADE agents responsible for receiving and pre-processing feeds from sensors. We call these agents *connectors*. Each *connector* collects data from a single sensor. Note that, since *connectors* being software

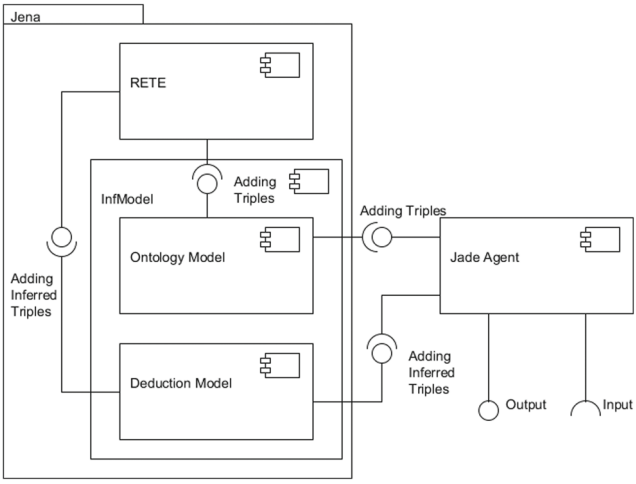

Fig. 4 The component diagram of the system

agents communicate by sending messages (this makes our proposed platform inherently discrete), sensors read continous data feeds. Thus, continuous acting components can be placed within the proposed framework. The process of communicating, with the specific sensor, is handled by an instance of a *Communication Module*. The *Communication Module* is an *interchangeable* module that processes raw sensor feed and forwards it to its *connector*. Note that in the previous version of the *Gateway*, there was only one *connector*, which interacted with the MAPS platform through the MAPS-specific engine (an instance of *Communication Module*). Since the SBC can handle multiple sources of information (such as *GPIO*, bluetooth, wi-fi, etc.), the *Gateway* should have at least one *connector* for each communication channel.

However, let us stress that "one *connector* per sensor approach" is a more natural one, since each connector can focus on one type of data and be easier replaced in case of a failure. A damaged connector that handles a single connection can be identified when it goes to a terminal state and destabilize the system with respect to only one data source. Because all *connectors* work independently from each other, we decided to introduce a control unit to the *Gateway*, which ensures stability of the resulting multi-agent system. The *GatewayMaster* agent is a JADE agent, which monitors a work-flow of *connectors*. This agent collects all preprocessed sensor feeds from the *connectors* and sends them to the decision making unit (in our case, the *GliderAgent* agent). If a malfunction is detected within the *Gateway* (for example, one of *connectors* "dies"), the *GatewayMaster* agent tries to fix the work-flow by rebooting the defective agent(s). If the system is in an "unrepairable state", the *GatewayMaster* agent informs the decision making unit about the situation.

It should now be obvious that the *Gateway* represents the first step to instantiate communication between heterogeneous IoT devices (platforms) and the Cyber part of the CPS. Each sensor (platform) could be associated with a separate *connector* agent that deals with the data feed. Note also that such *connector* agents can, if necessary, perform any required data (pre)processing to make it "understandable" to the Cyber part of the system. Finally, note that *connector* agents can act as gateways to IoT actuators, transmitting "decisions", undertaken by the Cyber, to the actuators residing in the IoT level of the CPS system.

5.2 Running GliderAgent on android

The following description applies to both Android Oreo and the newest version of JADE (4.5). The JADEAndroid is an add-on for the JADE agent platform, which allows to run a multi-agent system on the Android device. According to the documentation (JADE Android add-on Guide 2019), the JADEAndroid supports two modes of execution: (i) split-container, and (ii) stand-alone. The split-container execution mode requires a persistent (Internet) connection to a remote machine running the JADE main container. In this mode, the actual agent container is split between the mobile device (*front-end*) and the remote machine (*back-end*). The key drawback of this approach is the need to maintain the connection. If the connection is broken, then the agent container (and all its agents) is (are) suspended until the connection with the remote ("main") machine is re-established. The stand-alone execution mode, on the other hand, allows to run the full JADE agent platform on a mobile device. This ability is very important for applications similar to the *GliderAgent* that may not to be able to maintain the network connection with the remote host. Specifically, in the *GliderAgent* system the main container would have to be placed within the ground station. Since it is impossible to guarantee connection between the glider and a specific ground station (moreover, such connection would be highly undesirable because of resource consumption), we had to proceed with the stand-alone-mode-based implementation.

Let us start from a short description of how the JADEAndroid add-on works with the Android system. This description is crucial to understand the problems that had to be solved. In the Android system, there are two types of services: (i) *started* and (ii) *bound*. A bound service provides an interface, and an implementation of the *IBinder* class, through which other application components can communicate. Application components that are *bound* to the service (by calling the *bindService* method) are called *clients*. The JadeAndroid add-on contains two *bound services*, the *MicroruntimeService* and the *RuntimeService*. The *MicroruntimeService* wraps a split-container in the split-container execution mode. Through its interface, the *MicroruntimeBinder*, an application component can request to create or kill a JADE agent on a remote computer running the agent main container. The *RuntimeService*, unlike the *MicroruntimeService*, allows to initiate the main agent container and, therefore, creates a stand-alone agent platform within the Android system. In this case, the *bindService* method returns an instance of the *RuntimeServiceBinder* class.

Since the *RuntimeService* and the *MicroruntimeService* provide a way to manage the agent life-cycle in the JADE agent platform, there is another mechanism, which allows application components to interact with a JADE agent itself. The *Clients* (typically Android activities) communicate with the JADE agents via the Object-to-Agent (or shortly *O2A*) interface mechanism, introduced in JADE version 4.1.1. Each agent on the Android system should provide and implement an *O2A* interface. During the execution of the agent's *setup* method, such *O2A* interface should be registered by calling the *registerO2AInterface* method. Then, an application component can invoke the agent's methods, after obtaining its *O2A* interface. Moreover, if an agent enables the *O2A* communication, by calling the *setEnabledO2ACommunication*, its *clients* can send application-specific objects to an *O2A* queue by calling the *putO2AObject* method. In the case of the *GliderAgent*, the *O2A* interface mechanism is used to send the glider data for further processing. For instance, the *GliderAgent* receives the GPS feed and the sensor data feed, which are used to support pilots decision making; e.g. they can be used to detect life-threatening situations, like the possibility of oxygen deprivation. Here, the *GliderAgent* creates a warning for the pilot to put an oxygen mask on, if it finds out that the oxygen level dropped below the acceptable level.

Unfortunately, the JADEAndroid is not as easy to work with as the JADE itself. Above all, the official documentation for the add-on does not cover details of the stand-alone execution mode, which significantly differs from the split-container execution mode. Thus, the documentation does not provide best developer practices to solve common problems that may arise in Android projects that utilize the stand-alone approach. The most troublesome problem that we have encountered, was a difficulty in obtaining an *O2A* interface in the stand-alone execution mode. The *RuntimeService*, unlike the *MicroruntimeService*, does not provide a very useful method called *getAgent*, which returns an instance of the *AgentController* object. The *AgentContoller* class provides the *getO2AInterface* method. The way suggested by the source code of the JADEAndroid add-on, in the case of the stand-alone execution mode, is to invoke the *startAgent* methods, which returns an instance of the *AgentHandler* object to its callback function. However, the *AgentHandler* class does not provide the *getO2AInterface* method to classes outside of its package.

To address this problem, we have decided that the *GliderAgent* itself will provide its *O2A* interface via a custom *Application* class. The *Application* class is a base class, which allows to maintain the global application state. The Android system allows developers to provide their own implementation of the *Application* class by specifying its name in the *Manifest*. The instance of the *Application* class is accessible by all application components, which makes it the proper place to store commonly used data, such as an *O2A* interface.

5.3 Reasoning for *GliderAgent* agents

Infusing software agents with "intelligence" is a very interesting topic. There are multiple ways to achieve it. However, the nature of the *GliderAgent* system imposes a number of restrictions. Namely, (1) it is strongly preferred (though, not necessary) that the selected approach will be Java-based (for better integration with other components of the system). (2) It should run on mobile Android devices (as the *GliderAgent* agent does), which may have relatively limited resources (e.g. memory and/or battery). It will be also desirable that (3) the selected solution will be flexible enough to modify the "reasoning capabilities" of an agent (or a selected group of agents) without taking the whole system down (see, also Frackowiak et al. (2009) and Frackowiak et al. (2008)). Taking these limitations into account, we have considered three approaches to inserting "intelligence" into software agents. (A) Native methods provided by JADE, (B) rule-based expert systems, and (C) semantic technologies, including reasoners. While the detailed analysis can be found in Ganzha et al. (2014), let us summarize the most important (refreshed) facts relevant to the development of the *GliderAgent* system.

Let us start from a simple observation that JADE agents utilize *Java classes* to "store/represent their knowledge. "As a result, once compiled *Java classes cannot* be easily changed without altering the *entire* application. Thus, an agent has to be recompiled, each time when a change is introduced into its "knowledge." Furthermore, as discussed in Frackowiak et al. (2009) and Frackowiak et al. (2008), such recompilation may result in need to take down the whole system, to replace the old version of an agent (agents) with a new one. Therefore, despite the fact that the native method is sufficient for many scenarios, the *GliderAgent* agent, which has to operate in a constantly changing environment (cockpit of a glider), requires much more flexible solution. Namely, the *GliderAgent* 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 whole system). Note that this observation applies to any agent platform with "hard coded" knowledge and is of particular importance as the size and "gravity" of the (sub)system increases. Henceforth, it should be obvious that this solution is particularly unappealing for typical IoT scenarios.

Now, let us consider Rule-based Expert Systems (RES) that could be combined with JADE agents running on (possibly mobile) Android devices. First, note that many Java applications, and thus the RESs, are not natively compatible with Android. While having the same syntax and similar interfaces, there are key differences between the standard Java API and the Android API. (1) Android uses ART, a runtime environment written and maintained by Google, instead of the standard Java Virtual Machine. (2) Some popular Java packages are not included in the Android API. For example, the *javax* package (Swing, XML libraries, etc.) is missing, and has to be replaced by the Android's native classes. Therefore, to use a RES within JADE agents, the selected RES should be an active, open source project, with a small number of dependencies that can be partially rewritten/repacked for the Android API. Based on these assumptions, we have selected three possible candidates: 1) Roolie (2019), 2) Drools (2019) and 3) OpenL Tables (2019).

Today, three year after our initial investigation, there are still problems with these RESs (vis-a-vis our application area). Firstly, Roolie has not been updated since December 2013, and therefore, still does not provide an algorithm for matching rules, e.g. RETE. Furthermore, this likely means that the project has been abandoned. Secondly, Drools (version 7.24, updated in July 2019) while being a very robust software, reaches the size that poses a serious question about its usability on resource-limited mobile devices (placed on a glider with a restricted energy supply). Finally, OpenL Table (version 5.22.5, released in July 2019) uses

proprietary data formats (Excel and Word), which are not well-suited for an open source type system.

Eventually, to implement a "brain" for the *GliderAgent* agent, we have decided to utilize semantic technologies. They provide us with a flexible mechanism for maintaining agent knowledge and, by utilizing flexible and extensible ontologies, ensure proper information handling. Within an ontology, facts are stored in a form of triples (subject, predicate, object) i.e. *Glider's battery is equal to 10%* (here, *Glider's batter* is a subject, *is equal to* is a predicated and *10%* is an object). Applications can use a reasoner to infer logical consequences from them (i.e. *Glider's battery is equal to 10%*, so *Glider's battery is in power-safe mode*). 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.

We decided to use the Apache Jena for knowledge base manipulation. Jena is a well-documented, open source framework for building semantic web and Linked Data applications. 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.

We have found three community projects aiming at running Jena on Android. (1) The Androjena (2019) project supported only a subset of the Jena features and was discontinued in 2010. (2) The Apache Jena on Android (2019) project tried to fully integrate Jena (with all its features) with the Android OS. However, the latest, stable version of Apache Jena on Android was released for the outdated Jena (version 2.7.0), on August 7, 2012. This indicates that also this project has been abandoned for good. (3) Jena Android (2019) is the most recent project started in November 2014. Unlike other Jena ports for Android, it does not publish deployment-ready binaries, but provides Maven (Apache maven 2019) build files of the port. The collection of scripts only automate the entire process. While compiling, Maven is instructed to replace and repack all missing dependencies. Thanks to this, everyone receives a fully automated tool that can compile Jena and its dependencies. Moreover, this project works with the latest Jena (version 3.12.0, updated May 2019).

Taking into account that all three projects ware not updated for more than a year after their initial release, we decided to utilize Jena Android in the *GliderAgent* system. First of all, this project was the most promising because it does not depend on its own development cycle. Secondly, it is a collection of scripts that can be easily updated. Last but not least, we have successfully use this project in our previous work presented in detail in Ganzha et al. (2014). Specifically, we used Jena Android to implement an agent brain composed of a set of rules.

Our implementation of agent intelligence 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 (Simple logging facade for java 2019) and Xerces (Apache xerces 2019) are external projects. SLF4J (Simple Logging Facade for Java, 1.5.8 released in August 2009) serves as a simple abstraction for various logging frameworks. It allows the user to plug-in the desired logging framework at the deployment time. The SLF4J for Android can be downloaded from the SLF4J website. The Xerces (licensed to the Apache Software Foundation, 2.11.0 released in October 2014), is intended for creation and maintenance of XML parsers. Unfortunately, Xerces uses *javax.** namespace (which is not available in Android) and, thus, had to be rewritten. Lucky, Maven builds from Jena Android refractors all namespaces on-the-fly during compiling.

After addressing all technical difficulties, let us describe how all components come together. The component diagram is presented in Fig. 4. There, we can see two software components: (1) Jena and (2) 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. Within the "brain", the ontology is stored as an instance of the *InfModel* class. Next, the 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*. The RETE algorithm runs until there are no rules to fire.

To illustrate how the 'brain' operates, let us analyse a simplified example of an ontology and a set of rules presented in Listing 1. The ontology contains a set of fact about a battery on the glider. Each rule is composed of label, condition and inferred fact. For example, the *log critical* rule can be read as "if object *D* has a property *power* equal to *A* and *A* is equal to *Critical*, then *D* also have *loglevel* equal to *ErrorOnly*". The variables D and A will be equal to the object representing our glider (it is the only subject having the property *power*) and to its power level (a number between 0 and 100). JADE agent analyses its the environment (extracts facts). The *Glider agent* checks the battery power. Let us assume that the new value is equal to 4. During the first iteration, the "brain" will fire the *battery critical* rule, because the *Glider agent* modified the *power* property from its initial value of 100 to 4 (other conditions are not met). Thus, the "brain" will infer a new fact that the *battery* is equal to *Critical*. During the second iteration, the algorithm will fire the *log critical* rule

```
<!-- 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:power rdf:datatype="http://www.w3.org/2001/XMLSchema#int">100</eg
            :power>
        <eg:battery rdf:resource="urn:x-hp:eg/Normal"/>
        <eg:loglevel rdf:resource="urn:x-hp:eg/All"/>
    </rdf:Description>
</rdf:RDF>

<!-- RULES -->

[battery_normal: (?d eg:power ?a) ge(?a,20) lessThan(?a,100)
    -> (?d eg:battery eg:Normal)]
[battery_low: (?d eg:power ?a) ge(?a,5) lessThan(?a,20)
    -> (?d eg:battery eg:Low)]
[battery_critical: (?d eg:power ?a) ge(?a,0) lessThan(?a,5)
    -> (?d eg:battery eg:Critical)]

[log_critical: (?d eg:power eg:Critical)
    -> (?d eg:loglevel eg:ErrorOnly)]
```

Listing 1 Facts and rules used in the battery example

because the first iteration fulfilled its condition (the *power* is equal to *Critical*). In the third iteration, the algorithm will not fire any rule and stop execution.

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*. The *Glider agent* will be informed that the *power* property is now *Critical* and the *loglevel* is now *ErrorOnly*. 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.

Because all the reasoning takes place inside Jena, our "brain", similarly to the *Gateway* agent, can be easily run as a separate component in other projects. Namely, the set of facts and rules are independent from the implementation, so Jena can reason on any data and any rules. Moreover, since our components takes and returns triples, the 'brain' can be used without changes inside the source code.

It is worth to notice that the "brain" is an external component with respect to the JADE agent. Thus, this component can be easily replace or "improved", but, as long as it is understood by the agent, the structure of the system does not change. This may lead to an interesting dynamics between different agents using different "brains". For example, they can come up with their individual conclusions and, then, vote for the best one. As described in Stadnik et al. (2008) and Ganzha et al. (2010), combining multiple information sources (multiple "brains") can be beneficial providing a proper strategy is used.

6 Running *GliderAgent* 3.0

To test the *GliderAgent* 3.0, we have set up a test environment. The *GliderAgent* agent was installed on a Samsung Galaxy S7 Edge smart phone running Android 7.0 Nougat. The *Gateway* was deployed on the Banana Pi (A20 ARM Coretx -A7 Dual-Core, 1GB DDR3 RAM) running the latest Armbian (mainline kernel 4.14.y) (Bananian linux 2019) and OpenJDK 7 (1.7.0_79). To emulate sensors feed and to execute scenarios, we used a *GliderEmulator* (re-developed with the prototype). The *GliderEmulator* was deployed on a Dell laptop (Intel Core i5-4200U, 16GB RAM). Each device was connected to the same network. During the tests, we used the JADE platform in version 4.5 (released 2017-06-08).

In the first scenario, we tested the ability of the *Gateway* to detect and repair a critical state. Let us assume that the *GliderAgent* is flying near an airport. The *Gateway* receives sensors feed from the battery. Suddenly, the battery power level reaches a critical state and the remaining battery power is equal to 0%. However, after short time, the battery power level is back to a "normal state". The same situation repeats few more times. First, the *GatewayMaster* should detect a malfunction of the system (it is constantly switching between critical and normal states) and try to repair it by rebooting defective agents (in our case the battery *connector* agent). If the *connector* is still sending contradictory information, then the *GatewayMaster* should inform the *GliderAgent* agent about it.

In Fig. 5, we can see the output from the *GatewayMaster* and the JADE *AMS*. First, the battery *connector* agent is killed by the *GatewayMaster* and, then, it is recreated. However, the battery *connector* continuously sends contradicting messages. As a result, the *GatewayMaster* "kills" the battery *connector* agent once more, and informs the *GliderAgent* about it. Shortly after, a message is displayed on the smart phone screen that "battery sensor may be broken."

In the second scenario, we test the ability of the *GliderAgent* agent to interfere facts, based on different sources of information. The *GliderAgent* agent receives a GPS feed

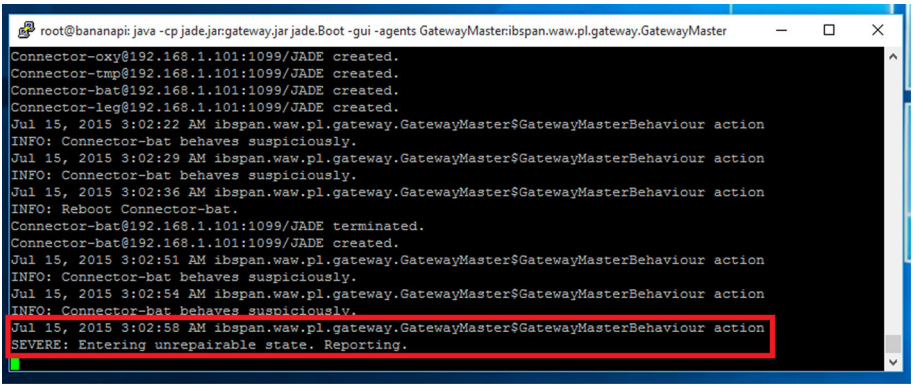

Fig. 5 Running the*GatewayMaster* on BananaPi

from the *GliderEmulator*. Moreover, it receives sensed data from the sensors. To represent sensed data, the system utilizes Semantic Sensor Network Ontology (or *SSN* in short) (Semantic sensor network ontology 2019). In Listing 2, the battery power level sensor (*BatteryPowerLevelSensor_1*) makes an observation (*PowerLevelObservation_0*). *PowerLevelObservation_0* states that the glider battery (*Battery_1*) has its power level (*PowerLevelMeasurementValue*) equal to 50%. Based on such information, the *GliderAgent* agent can optimize the power consumption with respect to the context. Furthermore, the system can communicate with other sensors which describe their data in *SSN*.

In Listing 3, we can see fragments of the glider ontology used to describe gliders and their status (geo-position, sensed data, parameters). The ontology was created based on our many years of experience with the *GliderAgent* system and was fine-tuned for the "brain" component (Ganzha et al. 2014). Following the ontology, the listing contains a set of rules that detect low batter level and low oxygen pressure. Each time the *GliderAgent* agent receives a geo-position or sensed data, the glider ontology is updated. This action can trigger execution of the RETE algorithm. Here, the "brain" interferes new facts based on many sources of information. For example, *log1* and *log2* determine the *GliderAgent* agent log level, where 0 stands for "all", 1 for "warning" and 2 for "critical". In the "Training" flight mode, everything is logged (the system does not care about the battery level; see, Section 2.3). However, in other modes (e.g. in/flying, in particular, see Section 2.2), if the battery level is below a threshold (in our case 0.15), no warning messages will be displayed. The *GliderAgent* is aware of its context and adjusts resources to its need. During the "Training" flight, the *GliderAgent* displays all notifications (the system assumes that the glider is being piloted by as inexperienced pilot who needs more information). For 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. Next, the *GliderAgent* agent starts to receive information generated by the *GliderEmulator*, which runs on PC and is connected to the same network. 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. Depending on the context (here, it is set to the "Training" flight) and the battery level, the *GliderAgent* agent decides to display a warning message. In this case,

```
<rdf:RDF
    (...)
    xmlns:ssn="http://purl.oclc.org/NET/ssnx/ssn#"
    (...)
  <owl:Ontology rdf:about="http://example.com/ssnx/sensor">
    (...)
    <owl:imports rdf:resource="http://purl.oclc.org/NET/ssnx/ssn"/>
    (...)
  </owl:Ontology>

  (...)

  <PowerLevelSensorOutput rdf:about="http://example.com/ssnx/sensor#
      PowerLevelSensorOutput_0">
    <ssn:isProducedBy rdf:datatype="java:org.apache.jena.ontology.impl.
        IndividualImpl">
       http://example.com/ssnx/sensor#BatteryPowerLevelSensor_1
    </ssn:isProducedBy>
    <ssn:hasValue rdf:datatype="java:org.apache.jena.ontology.impl.
        IndividualImpl">
       http://example.com/ssnx/sensor#PowerLevelMeasurementValue
    </ssn:hasValue>
  </PowerLevelSensorOutput>

  <rdf:Description rdf:about="http://example.com/ssnx/sensor#
      PowerLevelMeasurementValue">
    <hasQuantityValue rdf:datatype="http://www.w3.org/2001/XMLSchema#float">
       100.0
    </hasQuantityValue>
  </rdf:Description>

  <PowerLevelObservation rdf:about="http://example.com/ssnx/sensor#
      PowerLevelObservation_0">
    <ssn:observedBy>
      <BatterySensor rdf:about="http://example.com/ssnx/sensor#
          BatteryPowerLevelSensor_1">
        <ssn:onPlatform>
          <ssn:FeatureOfInterest rdf:about="http://example.com/ssnx/sensor#
              Battery_1">
            <rdf:type rdf:resource="http://purl.oclc.org/NET/ssnx/ssn#
                Platform"/>
          </ssn:FeatureOfInterest>
        </ssn:onPlatform>
      </BatterySensor>
    </ssn:observedBy>
    <ssn:observationResult>
      <PowerLevelSensorOutput rdf:about="http://example.com/ssnx/sensor#
          PowerLevelSensorOutput_0">
        <ssn:isProducedBy rdf:datatype="java:org.apache.jena.ontology.impl.
            IndividualImpl">
           http://example.com/ssnx/sensor#BatteryPowerLevelSensor_1
        </ssn:isProducedBy>
        <ssn:hasValue rdf:datatype="java:org.apache.jena.ontology.impl.
            IndividualImpl">
           http://example.com/ssnx/sensor#PowerLevelMeasurementValue
        </ssn:hasValue>
      </PowerLevelSensorOutput>
    </ssn:observationResult>
    <ssn:featureOfInterest rdf:resource="http://example.com/ssnx/sensor#
        Battery_1"/>
  </PowerLevelObservation>


</rdf:RDF>
```

Listing 2 Sensed data represented in Semantic Sensor Network Ontology

```
<!-- FACTS -->

<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:eg="http://ibspan.waw.pl.glider/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#" >
  <rdf:Description rdf:about="urn:x-hp:eg/Glider">
    (...)
    <eg:alt rdf:datatype="http://www.w3.org/2001/XMLSchema#double">0.0</eg:
        alt>
    <eg:lat rdf:datatype="http://www.w3.org/2001/XMLSchema#double">0.0</eg:
        lat>
    <eg:lng rdf:datatype="http://www.w3.org/2001/XMLSchema#double">0.0</eg:
        lng>
    (...)
    <eg:context rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
        Training</eg:context>
    (...)
    <eg:id rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</eg:id>
    (...)
    <eg:log rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</eg:log>
    (...)
    <eg:state rdf:resource="urn:x-hp:eg/Normal"/>
  </rdf:Description>
</rdf:RDF>

<!-- RULES -->

(...)
[log1: (?g eg:context ?c) equal(?c, "Training") -> (?g eg:log 0)]
[log2: (?g eg:context ?c) notEqual(?c, "Training") (?g eg:bat ?b) le(?b,0.15)
     -> (?g eg:log 2)]
(...)
[low_oxygen: (?g eg:altitude ?a) ge(?a,3000) lessThan(?a,4000) (?g eg:log ?l)
     ge(?l,0) -> (?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)]
(...)
```

Listing 3 Facts and rules used in the second scenario

the Cyber part of our system has successfully "combined" information from two different IoT devices and a context.

7 *GliderAgent* 3.0 as a general purpose cyber-physical system modeling platform

During the development of *GliderAgent* 3.0, we made three important observations. (1) The *GliderAgent* system is composed of interchangeable components, which can be modified or replaced depending on the context/use case scenario. (2) It has many features of a cyber-physical system (e.g. sensors, human-in-the-loop, intelligence and proactive behavior). (3) It features a possible solution to the problem of interconnecting heterogeneous IoT devices (and even platforms). Henceforth, let us now reflect how the *GliderAgent* system can be adapted to other scenarios.

Cyber-physical systems can be found in areas as diverse as aerospace, automotive, communication, infrastructure, energy, healthcare, manufacturing, transportation, robotics,

consumer appliances and so on. To create a general purpose cyber-physical system that can be easily adapt to a variety of different scenarios is not a simple tasks. Starting from the prototype of the *GliderAgent* system in 2011, we came a long way during which we upgraded the components from "glider system specific" to more general. As a result, the *GliderAgent* system has three main components: a) the *Gateway*, b) the "brain", and c) the emulator.

As shown in Section 5.1, and as proved experimentally, the *Gateway* can connect a JADE platform with "almost anything." In version 1.0 and 2.0, the *Gateway* was used to exchange messages between the JADE and MAPS platforms. Thanks to our experience with SunSpots (devices with very limited resources) and the WSN, we were able to create a highly-optimized and efficient tool. In version 3.0, the *Gateway* communicates with other non-JADE modules (i.e. Raspberry Pi, Banana Pi, PC) via *connector* agents and *Communication Modules*. Depending on the scenario, the *Gateway* could receive feeds from smart-meters via wi-fi or Bluetooth (to be used in "smart grid scenarios"), or health information from smart-bands (to be applied in "healthcare scenarios"). As a matter of fact, for plain curiosity we have managed to connect devices within our system using both Bluetooth and Wi-fi. Moreover, it can be run on other devices than a PC, such as single-board computers or smart-phones (using a slightly adapted JadeAndroid). Here, we have checked that the system can gather information from multiple sources via different communication channels. It is worth noting that the *Gateway* does not require significant modifications in its internal stucture (only *Communication Modules* are changed). In this way the *Gateway* can be seen as a generic middleware that provides a bridge between heterogeneous IoT devices (platforms) and the Cyber part of the CPS.

Thanks to our novel approach to infusing JADE agent with "intelligence", based on semantic technologies and a rule-based engine, the *GliderAgent* agent can be easily adapted to multiple contexts. The key to achieve this was obtained by separating the application logic from the compiled source code. In our approach, changes related to "reasoning," introduced to the system are reflected in the ontology (which can be reloaded at any moment) and, in most cases, should not require restating the entire application. This allows to update an agent knowledge base, and a set of rules, to be performed "on-the-fly.''

In the most "twisted" scenario, one can successfully replace the entire "brain" of an agent, as long as there are no need for changes in the source code. For example, the *GliderAgent* could instantly become a *SmartGridAgent* that displays warning/error messages about devices in the smart grid, based on the sensed information. The only operation that is actually needed is to replace the ontology and provide the new sensor feed. Of course, we assume that in both cases the same functions (including the text of the messages) are used to display messages on the mobile device, thus no new code is introduced. Obviously, this scenario is somewhat strange, but the main idea holds. Overall, the *GliderAgent* 3.0 became a general platform that allows one to easily instantiate multiple, different, cyber-physical systems.

It should be noted that the proposed approach is conceptually close to the way that the "world of travel" attempts at integrate available services using the Open Travel Alliance (OTA, Opentravel alliance (2019)) messaging standard. The OTA developed a set of XML-based travel-world describing messages that cover "all" key travel scenarios (e.g. searching for a hotel with given properties, and making reservations). Here, each player in the "world of travel" can use its own local data storage and processing mechanisms and, as long as it can send/receive/understand OTA messages, it can successfully communicate with other players. Similarly, in the *GliderAgent* system, heterogeneous sensor feeds are translated to formatted strings represented in the *common* ontology. Next, the reasoner is used to infer what action (if any) the system should undertake. Henceforth, adding an additional device

would require establishing a new *connector* agent (and *CommunicationModule* within the *Gateway*, and modification of the ontology and the rules.

8 Concluding remarks, future directions

In the paper we have described the developments that led to the implementation of the *GliderAgent* version 3.0. During this process, we have faced unique challenges that required a novel solutions that are descried in details in their respective sections. Naming a few, we can highlight: a) creating an interoperable and general-purpose gateway (Section 5.1) b) running agents on mobile devices (Section 5.2), c) providing an "inteligent" way to process userś input in MAS (Section 5.3).

Moreover, we have also shown, how this newest, modular, version of the *GliderAgent* can become a general purpose Cyber-Physical platform that, thanks to the utilization of a rule-based engine system, can be easily adapted to other CPS scenarios. Namely, the proposed "brain" for agents can contain any set of facts and rules. Furthermore, it was argued that the use of semantic processing simplifies dealing with the problem of combining multiple heterogeneous Internet of Things devices (sensors and actuators) within a single Cyber-Physical System.

Since our recent work focuses mainly on the "intelligent" part of the system, there is still an open question of how the *GliderAgent* system can cooperate with other projects like the *OGN* network. In this paper, we laid down the foundation for a discussion by pointing out similarities and differences between these two systems.

In the near future we plan to, first, apply the developed system to the smart grid scenario(s) to validate our claims that the *GliderAgent* is a general CPS platform. We also plan to initiate studies concerning scalability of the approach.

References

Gab, A., Andreou, P., Ganzha, M., Paprzycki, M. (2010). GliderAgent—a proposal for an agent-based glider pilot support system. In *2010 15th international conference on methods and models in automation and robotics (MMAR),* (pp. 55–60). https://doi.org/10.1109/MMAR.2010.5587263.

Domanski, J.J., Dziadkiewicz, R., Ganzha, M., Gab, A., Mesjasz, M.M., Paprzycki, M. (2012). Implementing glideragent—an agent-based decision support system for glider pilots. In: Software agents, agent systems and their applications, IEEE Press, pp. 222-244.

XCSoar (2019). http://www.xcsoar.org/.

Jennings, N., & Wooldridge, M. (2001). Agent-oriented software engineering, Handbook of Agent Technology.

Sun SPOT (2019). http://www.sunspotworld.com/.

Java agent development framework (2019). http://jade.tilab.com/.

Mesjasz, M., Cimadoro, D., Galzarano, S., Ganzha, M., Fortino, G., Paprzycki, M. (2013). Integrating jade and maps for the development of agent-based wsn applications. In *Intelligent distributed computing VI: Proceedings of the 6th international symposium on intelligent distributed computing - IDC 2012, Calabria, Italy, September 2012* (pp. 211–220). Berlin: Springer.

What is a raspberry pi (2019). https://www.raspberrypi.org/.

Banana pi – a highend single-board computer (2019). http://www.bananapi.org/.

Cyber-physical systems (2012). http://cyberphysicalsystems.org/.

IDEAS'2020: A Tour of Tomorrow's World (2019). http://www.ideen2020.de/en/2993/.

Internet of things towards ubiquitous and mobile computing (2019). http://research.microsoft.com/en-us/um/redmond/events/asiafacsum2010/presentations/guihai-chen_oct19.pdf.

Nwana, H.S., & Ndumu, D.T. (1999). A perspective on software agents research. *Knowl. Eng. Rev.*, *14*(2), 125–142. https://doi.org/10.1017/S0269888999142012.

Belecheanu, R., Munroe, S., Luck, M., Payne, T.R., Miller, T., McBurney, P., Pechoucek, M. (2006). Commercial applications of agents: lessons, experiences and challenges. In: 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), Hakodate, Japan, May 8-12, 2006, pp. 1549–1555. https://doi.org/10.1145/1160633.1160932.

Ganzha, M., & Lakhmi, J.C. (2009). *Multiagent systems and applications*. New York: A John Wiley and Sons, Ltd.

Airspace in the U.S., f.rom.t.he.F.'sA.eronautical.I.nformation.M.anual. (2019). http://www.faa.gov/air_traffic/publications/ATpubs/AIM/TOC.html (2010.02.04).

FLARM (2019). http://www.flarm.com/.

Altitude sickness (2019). http://www.altitude.org/altitude_sickness.php.

Soar (2019). http://www.aeroclub.student.kuleuven.ac.be/lvzc/soar/.

Seeyou mobile (2019). http://www.naviter.com/products/seeyou-mobile/.

WinPilot (2019). http://www.winpilot.com/.

Skylines (2019). https://skylines.aero/.

Atkins, E.M., Durfee, E.H., Shin, K.G. (1999). Autonomous flight with circa-ii. In *Autonomous Agents'99 workshop on autonomy control software*.

Hook, M., Romsey, H., Purcell, A., Watkin, R. (2007). Autonomous soaring. In *Autonomous systems, 2007 institution of engineering and technology conference*.

IDC: Analyze the Future (2019). http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

Android system (2019). http://www.android.com/.

Google (2019). http://www.google.com/intl/en/about/.

Apache license (2019). http://www.apache.org/licenses/.

Android lollipop (2019). https://www.android.com/versions/lollipop-5-0/.

Apple watch (2019). https://www.apple.com/watch/.

Android wearable (2019). https://developer.android.com/wear/.

Arduino (2019). https://www.arduino.cc/.

Mobile-c (2019). http://www.mobilec.org/.

The Foundation of Intelligent Physical Agents (FIPA) (2019). http://www.fipa.org/.

Jadex (2019). http://sourceforge.net/projects/jadex/.

Jason (2019). http://jason.sourceforge.net/.

Chmiel, K., Gawinecki, M., Kaczmarek, P., Szymczak, M., Paprzycki, M. (2005). Efficiency of jade agent platform. *Sci. Program.*, *13*(2), 159–172.

Aiello, F., Fortino, G., Gravina, R., Guerrieri, A. (2011). A java-based agent platform for programming wireless sensor networks. *Comput. J.*, *54*(3), 439–454.

Sun Small Programmable Object Technology (Sun SPOT) (2012). Documentation and software. http://www.sunspotworld.com.

Web Oriented JavaBeans and Swing (2019). http://java.net/projects/swingx-ws/.

OpenStreetMaps (2019). http://www.openstreetmap.org/.

Caire, G., & Pieri, F. (2011). LEAP User Guide. http://jade.tilab.com/doc/tutorials/LEAPUserGuide.pdf, Technical Report.

JADE Android add-on Guide (2019). http://jade.tilab.com/doc/tutorials/JADE_ANDROID_Guide.pdf.

Java se for arm (2019). http://www.oracle.com/technetwork/java/javase/downloads/jdk8-arm-downloads-2187472.html.

Frackowiak, G., Ganzha, M., Gawinecki, M., Paprzycki, M., Szymczak, M., Badica, C., Han, Y., Park, M. (2009). Adaptability in an agent-based virtual organization. *IJAOSE*, *3*(2/3), 188–211. https://doi.org/10.1504/IJAOSE.2009.023636.

Frackowiak, G., Ganzha, M., Paprzycki, M., Szymczak, M., Han, Y., Park, M. (2008). Adaptability in an agent-based virtual organization - towards implementation. In: Web information systems and technologies, 4th international conference, WEBIST 2008, Funchal, Madeira, Portugal, May 4-7, Revised Selected Papers, 2008, pp. 27–39. https://doi.org/10.1007/978-3-642-01344-7_3.

Ganzha, M., Mesjasz, M.M., Paprzycki, M., Ouedraogo, M. (2014). Inserting "brains" into software agents–preliminary considerations. In: Internet and distributed computing systems, Springer International Publishing, pp. 3–14.

ROOLIE – A Simple Java Rule Engine (2019). http://roolie.sourceforge.net/.

JBoss Tools – Drools (2019). http://tools.jboss.org/features/drools.html.

Openl tables (2019). http://openl-tablets.sourceforge.net/.

androjena – jena android porting (2019). https://code.google.com/p/androjena/.

Apache jena on android (2019). http://elite.polito.it/index.php/research/downloads/182-jena-on-android-download.

Apache jena for android (2019). https://github.com/seus-inf/jena-android.

Apache maven (2019). https://maven.apache.org/.
Simple logging facade for java (2019). http://www.slf4j.org/.
Apache xerces (2019). http://xerces.apache.org/.
Stadnik, J., Ganzha, M., Paprzycki, M. (2008). Are many heads better than one—on combining information from multiple internet sources . *Intel. Distr. Comput. Syst. Appl.*, *162*, 177–186.
Ganzha, M., Paprzycki, M., Stadnik, J. (2010). Combining information from multiple search engines-preliminary comparison. *Inf. Sci.*, *180*, 1908–1923.
Bananian linux (2019). https://www.armbian.com/.
Semantic sensor network ontology (2019). https://www.w3.org/2005/Incubator/ssn/ssnx/ssn/.
Opentravel alliance (2019). http://www.opentravel.org/.
Pintea, C.-M., Tripon, A.C., Avram, A., Crişan, G.-C. (2018). Multi-agents features on Android platforms. *Complex Adaptive Systems Modeling*, *6*(1), 1 0. https://doi.org/10.1186/s40294-018-0061-7.
Open Glider Network tracking protocol. Open Glider Network. http://wiki.glidernet.org/ogn-tracking-protocol.
Fortino, G., Gravina, R., Guerrieri, A., Di Fatta, G. (2013). Engineering Large-scale Body Area Networks Applications. In *Proceedings of the 8th International Conference on Body Area Networks* (pp. 363–369). ICST, Brussels: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). https://doi.org/10.4108/icst.bodynets.2013.253721.