Programming Intelligent IoT Systems with a Python-based Declarative Tool*

Fabio D'Urso, Carmelo Fabio Longo and Corrado Santoro

Dipartimento di Matematica e Informatica - University of Catania
Viale Andrea Doria, 6 - 95125 - Catania (ITALY)
{durso,santoro}@dmi.unict.it, fabio.longo@unict.it

Abstract. IoT applications are traditionally characterised by a set of interacting small devices equipped with microcontrollers (MCUs). Basically, they are often programmed in bare-metal using the C or C++ language; however, IoT applications are becoming more sophisticated thus including forms of autonomous behaviours that, in some cases, have the objective of presenting a certain degree of intelligence or reasoning; but since reasoners are traditionally designed using logic/declarative approaches, their implementation into embedded devices that use C/C++ as the main development language is no simple at all. Nevertheless, there are a fair number of porting of high-level languages to MCU platforms that could help to overcome the cited difficulties; and among them, *MicroPython* is one of the most interesting: it is a fully-featured Python environment running on a wide range of MCUs, also providing a small memory footprint and good performances. On this basis, this paper presents a Python framework called PHIDIAS that allows the development of logic/declarative code seamless running into a Python program. PHIDIAS is able to give Python programs the ability of performing logic-based reasoning (in the Prolog style), also letting developers to write *reactive procedures*, i.e. pieces of program that can promptly respond to environment events, which represent a typical case in IoT applications. PHIDIAS also includes a library for the interfacing of I/O peripherals of a microcontroller. The paper presents the PHIDIAS framework, highlighting features and advantages, and also provides a case-study in order to assess the effectiveness of the proposed solution.

Keywords: IoT · Multi-agent systems · MicroPython · Logic/declarative programming

1 Introduction

In the last few years, the development of the Internet-of-Things has led to the introduction, in the market, of a wide number of small devices featuring very different functionalities but able to interact to each other, and to other kind of

systems, with the objective of helping us in day-to-day activities [12]. As just some examples, many car models are "smart": they are still tightly integrated with our mobile phones, and it is expected, in the near future, that they will be able to interact with other cars thus improving driver comfort and safety, or reducing travel times, fuel consumption, etc. In a similar way, our homes are becoming full of *smart objects*, starting from smart switches, smart lamps, surveillance cameras, up to vocal assistants, such as Amazon Alexa or Google Home, that allow occupants to control the whole home by means of voice commands.

Such devices are usually equipped with microcontroller units (MCUs), and include, apart from proper sensors and/or actuators, which are relevant to the specific features of the device itself, a connectivity module i.e. a Wi-Fi or Bluetooth chip for the interaction with other devices. From the software development point of view, these devices are in general programmed using the C or C++ language, with a development environment that often includes some libraries providing an abstraction layer for MCU peripherals; the developed firmware runs on bare-metal and only in very few cases an operating system (which is usually a small real-time executive) is included.

The current generation of such IoT devices are limited in functionalities, that is, smart sockets or smart switches have usually the capability of receiving or transmitting the on/off command, and sometimes include also measuring sensor acquiring data such as current or power, which may be useful to monitor energy consumption. But it is expected that, in the very near future, such devices will have the ability to exhibit a certain form of "intelligence" thus performing, in autonomy, tasks that could derive from a reasoning process. As an example, a smart lamp could learn home occupants habits and adapt off/on times or light intensity from what people are used to do; or a heating/cooling system could save a considerable energy by adapting its behaviour using data coming from proximity sensors, in order to turn on/off at the proper time, making the environment warm or cool, on the basis of human presence in the house.

Including intelligence in a smart device implies to add, to the firmware, a proper support for the AI techniques needed in the specific project, which could range from first-order logic inference, to more complex reasoning, up to machine learning-based approaches. In particular, when a form of reasoning is required, state-of-the-art techniques consider the use of *logic/declarative frameworks or languages* which, in turn, are traditionally based on Prolog or Prolog-like environments; the bad news is that such runtimes are usually not able to execute on devices like MCUs that feature a limited computational power and available memory.

However, we must remind that, in the field of high-level programming languages/platforms for MCUs, the *MicroPython* project has gained a very high interest, since it provides a Python VM running over a wide range of MCUs, featuring very interesting run-time performances and a limited size memory footprint. With these aspects in mind, the authors investigated about the possibility of porting a Python-based logic/declarative framework to MicroPython, thus al-

lowing developers to include reasoners in IoT devices, making them "more intelligent". The developed tool, called PHIDIAS, is able to support logic-based/BDI multi-agent systems in Python. PHDIAS is the evolution of PROFETA [11] and has the aim of providing, in a single framework, a knowledge system, a procedural system and a reactive system: altogether can be used to write *rational agents* that can not only react to events but also plan actions in order to proactively reach their goals. This paper describes the tool, reporting the syntax of the declarative language highlighting, in particular, the components that are specifically written for the MCU and that provide access to I/O peripherals and wireless communication modules. A case-study is also included, in order to let the reader understand the capabilities of PHIDIAS and its advantages in the context of intelligent IoT.

The paper is structured as follows. Section 2 deals with related work. Section 3 presents an overview of PHIDIAS. Section 4 describes the components provided by PHIIAS for a MCU scenario. Section 5 describes a case-study of a domotic application. Section 6 concludes the paper.

2 Related Work

The research on IoT intelligent systems reports a wide number papers, dealing with different scenarios and solutions.

The author of [16] provides a global conceptual overview of the synergy of AI and IoT with emphasis on its application in robotics and automation; the paper also outlines a number of case studies (home automation, oil-field production, smart robotics, smart manufacturing, and smart factory).

In [13] the authors present an architecture based on a distributed edge/cloud paradigm, which aims to let drones recognize objects during their flight, in order to balance benefits and cost of processing data at the edge, versus a central location. The reasoning models are trained in the cloud, then deployed as Docker containers and loaded into a shared repository, from which can be accessed by the edge components.

In [15], the authors deal with classrooms under-utilization in a real university campus, by instrumenting classrooms with IoT sensors to measure real-time usage, using regression learning algorithms to predict attendance, and performing optimal allocation of rooms to courses so as to minimize space wastage.

The authors of [17] propose an approach to decompose a complex AI application into simplified distributed modules connected by using the IoT technology. The framework, called *AItalk*, allows a developer to easily add, to existing IoT applications, AI mechanisms such as regression models, automatic data collection, real-time prediction, model training, etc.

In the context of run-time executive or programming languages for MCU devices, there are indeed few solutions. In general, since MCUs are programmed in C, any C-based run-time is theoretically able to execute onto such platforms, unless the memory footprint of the resulting code and the amount of RAM required is too much for the target device. As for C-based platforms, CLIPS [1] is

a run-time that supports writing and execution of ruled-based production systems by means of a LISP-like programming language and a data model based on "facts"; however, CLIPS, to run onto a MCU, must be properly ported by patching some system-dependent parts (like those relevant to mass storage) and, while this work seems feasible, at the present the authors do not have a knowledge of such a porting project. In the context of symbolic languages (also with functional/declarative philosophy), there are several implementations of LISP/Scheme; as just an example, uLISP [4] is porting of the language for many MCU-based boards, like Arduino, ESP8266, STM32, etc.; it is interesting since it has a small memory footprint and, since it is a standard LISP implementation, could serve as run-time environment for a symbolic reasoner or production system. As for imperative and object-oriented approaches, it's worth of notice the solution provided for the Java world which consists in the JavaME [3], a JVM profile, with a suitable class library, specifically designed to fit small/embedded environment; the solution is interesting, but (as reported on the web page) it is able to run only in high-end MCU devices (e.g. Cortex-M7, ARM11). Some hardware platforms, like the NodeMCU board [5], offer a Lua-based environment; while the solution is interesting, it suffers of the problem of a low spreading of Lua which, at the present, cannot be considered a mainstream language. A solution to run JavaScript on MCUs is provided by the Espruino project; it is a commercial project by a company which produces and sells STM32-based hardware platforms, also providing an open-source development environment that includes a JavaScript interpreter running on several MCUs [2]; the interpreter is able to run also on low-end devices and it will be one of the solutions that we will analyse in our future work. The Python-based solution is the one considered in this paper; the choice of this language relies on three main factors; first, the increasing popularity of the language itself, which is ranked first in 2019 [7]; second, the availability of our Python-based BDI system PROFETA [11]; and third the availability of MicroPython as an effective Python run-time for the majority of MCU-based solutions.

3 Overview of the PHIDIAS Platform

PHIDIAS [6] is a multi-agent platform that let developers write multi-agent systems using the Python language and the BDI paradigm. PHIDIAS[1] is the evolution of PROFETA [11, 8] and is a knowledge-based system supporting Prolog-like inference and allowing the implementation of agent behaviour by means of reactive and proactive rules. Like PROFETA, PHIDIAS provides a declarative language that offers logic/declarative constructs that can be mixed inside Python code; this is made possible by exploiting the object-oriented features of Python, in particular operator overloading. With respect to PROFETA, PHIDIAS adds the possibility of expressing Prolog-like goals, the support for multi-agent environments, the interaction among agents via messages (also belonging to different

[1] PHIDIAS is the acronym of *PytHon Interactive Declarative Intelligent Agent System*

machines), the possibility of tying procedures to events (to combine proactivity and reactivity), and some modifications to the declarative language syntax in order to make it simpler and more flexible. We report here an overview of PHIDIAS; readers interested in more details can refer to [6, 11, 10, 9].

3.1 Basics of PHIDIAS

Like any knowledge-based system, a PHIDIAS program is made of a data part, which is represented by several *beliefs*, properly defined by the developer according to application requirements, and a computational part made of a set of *rules*. The data part may also include Prolog predicates that, applied to the beliefs asserted in the knowledge base, can be used to derive new knowledge.

As for computational part, the rules that constitute the agent program can be of three types:

- **reactive rules**, they are executed as triggered by the occurrence of certain events, such as asserting a belief into or retracting a belief from the knowledge base;
- **procedural rules**, they are executed when specifically invoked, as in classical procedural system;
- **event procedures**, they combine both the types above, i.e. they are procedures that can invoked but wait for the occurrence of specified events in order to proceed with the execution of the relevant body.

Each rule has a *head*, that can be either *(i)* the specification of an event occurring when a belief is asserted or retracted, or *(ii)* the specification of a procedure that can be properly invoked. Optionally, a rule can have a *context condition* part (which is syntactically indicated by the symbol "`/`", i.e. *subject to*) specifying a predicate that must be true in order to execute the triggered rule itself; the predicate is a condition on values of parameters and/or the presence of one or more given beliefs in the knowledge base. The last part of a rule, which is syntactically specified with the "`>>`" symbol, is the list of actions that expresses the computation to be executed following the activation of the rule; an action may be the asserting/retracting a belief, invoking a PHIDIAS procedure, or invoking a *user-defined action* that is instead implemented in pure Python.

In order to let the reader understand how PHIDIAS works, we describe, in the following, two toy examples with the objective of providing the reader with the basics of PHIDIAS syntax and semantics.

The first example, the listing of which is reported in Figure 1, is a simple knowledge-based application: here, we are representing a world in which there are "students" that, sooner or later, become "graduated"; the rules of the program have the objective of keeping the knowledge consistent. As the figure shows, the first part of a PHIDIAS program is devoted to concepts and symbol declaration, i.e. beliefs, which are declared as subclasses of `Belief` (lines 5-6), and variables used in rules that must be declared using the statement `def_vars` (line 7). Referring to the example, the student state, for a person `X`, is represented by

```
1  from phidias.Types import *
2  from phidias.Lib import *
3  from phidias.Main import *
4
5  class student(Belief): pass
6  class graduated(Belief): pass
7  def_vars("X")
8
9  +graduated(X) / student(X) >> \
10         [ show_line(X, " is now graduated!"), -student(X) ]
11 +graduated(X) >> \
12         [ show_line(X, " is not a student"), -graduated(X) ]
13 +student(X) / graduated(X) >> \
14         [ show_line(X, " is graduated and cannot be a student again"),
15           -student(X) ]
16
17 PHIDIAS.run()
```

Fig. 1. An Example of a Reactive PHIDIAS Program

the presence of the belief `student(X)` in the knowledge base; in a similar way, the action of becoming graduated is represented by the assertion of the belief `graduated(X)`, but this is possible only if `X` is a "student": if this is the case, belief `student(X)` must be retracted. This is represented by three reactive rules reported in Figure 1, respectively in lines 9–10, lines 11–12, and lines 13–15. The first rule states that, as soon as the belief `graduated(X)` (with X any) is asserted (the "+" symbol means that something has been added to the knowledge base), if the belief `student(X)` (with **that** X) is already present in the knowledge base, then the following actions are executed: first something is shown on the console and then belief `student(X)` is removed (since X is no longer a student). The second rule has the same triggering event of the first rule but it is evaluated only if the prevision rule does not match (i.e. the belief `student(X)` is not in the knowledge base) and the result is an error message and the immediate removal of belief `graduated(X)`: indeed the sense is that it cannot exist a "graduated" that has not been previously a "student". The third rule is instead used to state that, once a student has been graduated, s/he cannot become once again a student; to this aim, the rule is triggered by the assertion of belief `student(X)` and, if the knowledge base already contains the belief `graduated(X)`, an error message is printed and the former belief is deleted.

The second example is given in Figure 2 which reports the listing of a PHIDIAS program that uses procedures. Here we are supposing that this program is controlling a robot having the task of repetitively finding and grabbing balls that are present in the environment. We are using the belief `BallPos(X,Y)` to keep track of the position of (known) balls; the procedure `CatchBall()`, in lines 12–13, has, as contextual condition, the presence of (any) belief `BallPos(X,Y)` in the knowledge base; if this is true (variables X and Y will be bound to the relevant values), the robot is driven towards ball's position (action `GoTo(X,Y)`), then the ball is picked (action `GrabBall()`) and the belief is removed from the knowledge base since that ball is no more present in the environment; finally,

```
1  from phidias.Types import *
2  from phidias.Lib import *
3  from phidias.Main import *
4
5  class BallPos(Belief): pass
6  class CatchBall(Procedure): pass
7  class FindBall(Procedure): pass
8  class Goto(Action): ...
9  class GrabBall(Action): ...
10 def_vars("X","Y")
11
12 CatchBall() / BallPos(X,Y) >> [ GoTo(X,Y), GrabBall(),
13                                  -BallPos(X,Y), CatchBall() ]
14 CatchBall() >> [ FindBall(), CatchBall() ]
15 FindBall() >> # ... activate computer vision system to find for the ball
16
17 PHIDIAS.run(main=CatchBall())
```

Fig. 2. An Example of a Procedural PHIDIAS Program

the procedure `CatchBall()` is called recursively in order to let the robot pick the next ball. On the other hand, when the `CatchBall()` is called and no ball position is known (i.e. no belief `BallPos(X,Y)` is present, line 14), the procedure `FindBall()` is first involved and then `CatchBall()` is recursively called; here the `FindBall()` (which is not detailed in the listing) has the objective of activating a computer vision system which should search for the ball and then assert the relevant `BallPos(X,Y)` belief in the knowledge base.

Apart from the functionality explained, the listing shows some important syntactical elements of PHIDIAS: beliefs, procedures and variable must be declared (lines 5–10), as it has been shown also in the first example, while `Action`s are basic elements that represents atomic computations that a PHIDIAS program can execute; in particular, they contains pure Python code (which is not shown in the figure) and constitute one of the pieces of the bridge that connects the PHIDIAS world with the Python world. Indeed, the way in which a PHIDIAS program interacts with the external world is supported by means of the two basic abstractions: *Beliefs* and *Actions*, which are described in the following.

Beliefs represent *input data* and thus are generated by (Python) computations that, by means of interaction with suitable software drivers, are able to gather/poll data; beliefs, in turn, can be used to within PHIDIAS rules, as triggers or simply data knowledge for procedures. In order to support data polling, and consequent belief generation, PHIDIAS provides a further abstraction represented by the `Sensor` class: any piece of user software in charge of generating beliefs can be encapsulated in a `Sensor` sub-class, in particular in its `sense()` method; this code is executed by the PHIDIAS runtime in parallel to other activities[2], and thus can feature blocking calls and/or periodicity. In a similar way, any kind of output activity of a PHIDIAS program must be encapsulated into a user-defined `Action` (sub-)class and, in detail, in its `execute()` method that

[2] each sensor has its own thread of execution

can also receive the proper parameters from the call performed within the body of a PHIDIAS rule.

```
1  from phidias.Types import *
2  from phidias.Lib import *
3  from phidias.Main import *
4
5  class a_belief(Belief): pass
6  class go(Procedure): pass
7
8  class Sender(Agent):
9     def main(self):
10       go() >> [ +a_belief()[{"to":"Receiver@dest-machine.mydomain"}] ]
11
12 Sender().start()
13 PHIDIAS.run_net(globals(),'http')
```

```
1  from phidias.Types import *
2  from phidias.Lib import *
3  from phidias.Main import *
4
5  class a_belief(Belief): pass
6  def_vars("X")
7
8  class Receiver(Agent):
9     def main(self):
10       +a_belief()[{"from":X}] >> [ show_line("Received a belief from ", X) ]
11
12 Receiver().start()
13 PHIDIAS.run_net(globals(),'http')
```

Fig. 3. Example of Two Communicating Agents in PHIDIAS

3.2 Multi-agent Systems in PHIDIAS

One key aspect that distinguishes PHIDIAS from its predecessor, PROFETA, is the support for multi-agent systems. In PHIDIAS, a developer can define several agents, each one with its own set of rules, that are able to interact to each other; agents can coexist within a single PHIDIAS runtime environment, running concurrently, or execute in different environment/computer and thus interact by means of a communication network.

Interaction among agents is performed by means of a message-passing mechanism that allows a sender agent to *assert a belief* in the knowledge base of another agent; using this mechanism, the belief is transferred to the destination agent that, in turn, can react by using a PHIDIAS rule triggered by the assertion of that belief.

From the syntactical point of view, as Figure 3 shows, sending a message implies to use the belief assertion statement which is annotated with a "to" tag that specifies the agent destination of the belief itself (see line 10 of Figure 3); addressing is performed using a classical naming scheme such as "localname @

machine-name-or-ip". In a similar way (see the bottom side of Figure 3 and in particular line 10), by annotating a belief assertion trigger event with the "from" tag, the name/address of the sender agent (if any) can be retrieved and used properly.

As for the underlying messaging mechanism, as the listings suggest (see line 13 of both listings in Figure 3), message transfer is performed by using HTTP: each PHIDIAS environment runs a HTTP server that receives beliefs properly encoded, identifies the destination agent and, if it exists, put the data into the relevant knowledge base, activating rules if any.

4 PHIDIAS for MCU Environments

In the world of IoT and Smart Objects, a key aspect is the way in which data are received from and sent to the environment; this may include also data gathering from sensors (e.g. presence sensor) or actuator driving (e.g. driving a lamp), or commands/data related to the interaction with other devices. From the software point of view, such an interface has the task of providing an access to the I/O peripherals of the MCU, such as GPIO, timers, communication interfaces, etc., allowing a developer to use the proper abstraction which are related to the language/platform used to implement the application. To this aim, MicroPython includes some modules that provide a suitable software interface to MCU peripherals, in particular:

- General-Purpose Digital I/O lines;
- PWM generation;
- Analog-to-Digital Converters;
- UARTs to support serial communication;
- Serial-Peripheral-Interface (SPI) and I^2C Bus.

It's up to the developer to use such modules to integrate a specific sensor that could use e.g. ADC or I^2C interface, or one or more actuators attached to e.g. a PWM line or UART or another communication interface. This is the task of the porting of PHIDIAS for MicroPython: it includes a Python package, called *phidiasmcu*, that provides a set of services able to the perform a bridge between the PHIDIAS world and the Python/MCU world. In the current implementation, the services offered are GPIO, ADC and communication via TCP/IP over WiFi.

GPIO services can be used to write an output line, to read an input line or to activate a notification when an edge is detected on an input line. To this aim, the API provided in PHIDIAS includes the following types:

- `OutPut(pin,value)`, it is a PHIDIAS action able to set the given line to a certain value (0 or 1);
- `InPin(pin,variable)`, it is a special PHIDIAS belief[3] that binds the variable to the actual value of the given pin;

[3] it is called `ActiveBelief`.

– `HandleInputPin(pin,edge)`, it is an action able to activate a trigger on the given pin according to the occurrence of a specified edge (either rising or falling); when the edge is detected the special belief[4] `PinEvent(pin,value)` is automatically generated, which can thus be used to trigger a PHIDIAS rule.

ADC services allow PHIDIAS programs to configure the analog-to-digital converter peripheral of the MCU, start conversions and retrieve obtained values. Two PHIDIAS types are available for this:

– `ADCSetup(pin)`, it is an action that can be used to configure the given pin as ADC input;
– `ADCRead(pin,variable)`, it is a special construct that (according to the way in which it is used) can act as a belief or an action; if it appears in the context condition of a rule, it acts as an (active) belief; if it is present in the body of a rule, it acts as an action; in both cases, the objective is to bind the variable to the ADC values sampled from the pin.

As for *communication services*, there are several aspects that must be taken into account when a PHIDIAS program runs onto a MCU. Basically, MCU systems do not possess the same networking/interconnection capabilities of a classical computer system; indeed, communication mechanisms of MCUs are quite limited in characteristics and range, and include systems typical of the IoT world, like Bluetooth Low Energy, 6LowPan, ZigBee, etc. In some cases, it is possible to use small Wi-Fi modules that, while implementing the whole TCP/IP stack, present some limitations, like the ability to act only as client or the possibility of opening only one connection.

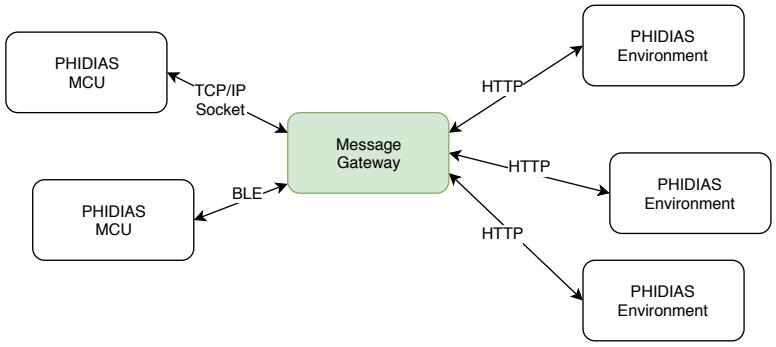

Fig. 4. Message Gateway

[4] it is a `Reactor`, i.e. a special belief that can generate a rule triggering event but is not added to the knowledge base.

Given this, in porting PHIDIAS to a MCU environment, we had to rethink the communication aspects since the HTTP-based interaction, "as-is", cannot be supported. But, at the same time, any other mechanism must not affect the syntax and the semantics of communication—from the language point of view—neither the naming scheme used for agent addressing. To this aim, we designed an architecture for the communication system for PHIDIAS in IoT environments, depicted in Figure 4, which is based on the presence of one or more *Message Gateways*, i.e. machines able to run a service that, on one side, exposes the HTTP protocol, behaving as a "classical" PHIDIAS environment and, on the other side, presents a connection endpoint that uses a specific protocol for the IoT environment. While the HTTP side is always the same for any kind of gateway, the other side depends on the protocol employed which, in turn, is tied to the hardware modules available on the IoT devices. In our current implementation, as it will be described in the case-study, we designed a *Socket-based Messaging Gateway* which is intended to be used with those Wi-Fi modules, often employed in MCU environments[5], that are limited to open only one TCP socket. In this case, a node running PHIDIAS connects to that gateway and handles message sending/reception through that single socket: it's up to the gateway to interpret specified recipients and deliver the message to the right node/agent addressed.

```
1  from phidias.Types import *
2  from phidias.Lib import *
3  from phidias.Main import *
4  from phdiasmcu.gpio import *
5  from phdiasmcu.idw01m1 import IDW01M1
6
7  class presence(Reactor): pass
8  class setup(Procedure): pass
9
10 class PresenceSensor(Agent):
11    def main(self):
12       setup() >> [ HandleInputPin("A1", "falling") ]
13       +PinEvent("A1",0) >> [ +presence()[{"to":"lighter@livingroom-bulb"}] ]
14
15 sock = IDW01M1() # Wifi module driver
16 sock.open(baud=57600) # Connect to the WiFi network
17 sock.wait_wifi_up()
18 sock.connect('gateway.address', 9999) # Connect to the Message Gateway
19 sock.send(b'livingroom-sensor-1\n') # announce node name
20
21 PresenceSensor().start(setup())
22 PHIDIAS.run_net(globals(), 'gateway', sock)
```

Fig. 5. The PHIDIAS Code of the Presence Sensor

[5] In particular, we are talking about the SPWF01SA Wi-Fi module [14].

5 Case-Study

In this Section we report a case-study which shows how to leverage PHIDIAS for the implementation of automated actions involving IoT devices in a domotic environment; it is a simple (but working) example, which however shows all the features of PHIDIAS in MCU/IoT environments. We focused on a system of smart light bulbs, grouped in rooms, where each rooms is equipped with one or more proximity sensor; the idea is to keep the lights on, according to human presence in a room, only for the needed time useful for home's inhabitants, in the view of energy saving. We consider that each sensor and bulbs are small MCU devices, equipped with a Wi-Fi module, some digital I/Os and running, of course, MicroPython and PHIDIAS; here, sensors send, via PHIDIAS messaging, on/off commands to the bulbs which, in turn, implement a proper timer to switch off the light when no more people are in the room.

```
1  from phidias.Types import *
2  from phidias.Lib import *
3  from phidias.Main import *
4  from phdiasmcu.gpio import *
5  from phdiasmcu.idw01m1 import IDW01M1
6
7  class presence(Reactor): pass
8  class my_timer(Timer): pass
9
10 class lighter(Agent):
11    def main(self):
12       +presence() >> [ OutPin("B0", 1), my_timer(60000).start ]
13       +timeout("my_timer") >> [ OutPut("B0", 0) ]
14
15 sock = IDW01M1() # Wi-fi module driver
16 sock.open(baud=57600) # Connect to the Wi-Fi network
17 sock.wait_wifi_up()
18 sock.connect('gateway.address', 9999) # Connect to the Message Gateway
19 sock.send(b'livingroom-bulb\n') # announce node name
20
21 ligher().start()
22 PHIDIAS.run_net(globals(), 'gateway', sock)
```

Fig. 6. The PHIDIAS Code of the Smart Bulb

Figure 5 shows the code running in each sensor of the room. Here we are considering that the hardware device that detects people presence has a digital interface and is connected to the input pin called "A1" of the MCU: the line is normally set to logic "1" and each time a movement is detected in the room, it goes to "0" keeping that state until a rest of at least 3 seconds is identified. On this basis, we have an agent called `PresenceSensor` that detects the edge generated by the sensor and sends a proper message to the bulb; in particular, the `setup()` procedure (line 12) activates a trigger on that line when a falling edge is detected: if this is the case, the `PinEvent()` belief is asserted and the rule in line 13 is triggered thus provoking a message sending to the smart bulb;

here, the destination agent specified is `ligher@livingroom-bulb`, i.e. the agent named "ligher" running the node named "livingroom-bulb". Lines 15–19 report the code needed to perform connection to the messaging gateway: here first the driver of the Wi-Fi modules is created which waits for the network, then performs connection to the gateway and announces its node name ("livingroom-sensor-1"). Lines 21 and 22 start the agent and the PHIDIAS runtime.

On the other side, the smart bulb runs the PHIDIAS code reported in Figure 6. Here we are supposing that the lamp is connected to the digital output line named "B0"; apart from the libraries used to drive digital outputs, this example uses timers which are natively provided by the PHIDIAS library. The startup code (lines 15–22) is the same to that of the sensor while the agent, named "ligther", has only two reactive rules: the former, line 12, is executed when the `presence()` belief is asserted: it is indeed activated to the message sent by the presence sensor; the body of the rule turns on the lamp through action `OutPin()` and starts a timer for 60 seconds; if the timer elapses the `timeout()` belief is generated and rule in line 13 is triggered, thus turning off the lamp; but if another `presence()` belief is received, due to activation of the presence sensor, the timer is restarted.

6 Conclusions

This paper described the MicroPython implementation of PHIDIAS, a BDI system that lets a developer write logic-based intelligent multi-agent system. The use of MicroPython allows the tool to run onto MCU-based hardware platform thus making it ready for the Internet-of-Things world. The implementation includes a set of libraries that provide suitable software abstractions for the various peripherals that usually are present in a MCU, as well as a messaging middleware to let agents interoperate transparently and independently of the running platform, i.e. MCU system or classical CPU system. A case-study has shown the basic capabilities of PHIDIAS that are provided for a MCU/IoT environment.

Our future work aims at writing more intelligent IoT application, in order to assess the characteristics of the tool for the specific context, and write additional libraries and drivers for other peripherals and/or sensors and actuators.

References

1. C language integrated production system (2017), http://clipsrules.sourceforge.net/
2. Espruino javascript interpreter source code (2019), https://github.com/espruino/Espruino
3. Java micro edition (2019), https://www.oracle.com/java/technologies/javameoverview.html
4. Lisp for microcontrollers (2019), http://www.ulisp.com/
5. Nodemcu (2019), https://www.nodemcu.com/index_en.html
6. Phidias web page (2019), https://github.com/corradosantoro/phidias
7. Popularity of programming language (2019), http://pypl.github.io/PYPL.html
8. Profeta web page (2019), https://github.com/corradosantoro/profeta

9. Fichera, L., Marletta, D., Nicosia, V., Santoro, C.: A methodology to extend imperative languages with agentspeak declarative constructs. In: Proceedings of the 11th WOA 2010 Workshop, Dagli Oggetti Agli Agenti, Rimini, Italy, September 5-7 (2010)
10. Fichera, L., Marletta, D., Nicosia, V., Santoro, C.: Flexible robot strategy design using belief-desire-intention model. In: Research and Education in Robotics-EUROBOT 2010, pp. 57–71. Springer (2011)
11. Fichera, L., Messina, F., Pappalardo, G., Santoro, C.: A Python Framework for Programming Autonomous Robots Using a Declarative Approach. Sci. Comput. Program. **139**, 36–55 (2017). https://doi.org/10.1016/j.scico.2017.01.003, https://doi.org/10.1016/j.scico.2017.01.003
12. Savaglio, C., Ganzha, M., Paprzycki, M., Bdic, C., Ivanovi, M., Fortino, G.: Agent-based internet of things: State-of-the-art and research challenges. Future Generation Computer Systems pp. 1038–1053 (2020)
13. Seraphin B. Calo, Maroun Touna, D.C.V.A.C.: Edge Computing Architecture for applying AI to IoT. In: IEEE International Conference on Big Data (BIGDATA). IEEE (2017)
14. STMicroelectronics: Command set reference guide for "AT full stack" for SPWF01Sx series of Wi-Fi modules. WWW (2018)
15. Thanchanok Sutjarittham, Hassan Habibi Gharakheili, S.S.K., Sivaraman, V.: Experiences With IoT and AI in a Smart Campus for Optimizing Classroom Usage. In: IEEE INTERNET OF THINGS JOURNAL, VOL. 6, NO. 5, OCTOBER 2019. IEEE (2019)
16. Tzafestas, S.G.: Synergy of IoT and AI in Modern Society: The Robotics and Automation Case. Robot Autom Eng J. 2018; 3(5): 555621. (2018)
17. Yun-Wei Lin1, Yi-Bing Lin1, C.Y.L.: AItalk: a tutorial to implement AI as IoT devices. IET Netw., 2019, Vol. 8 Iss. 3, pp. 195-202 (2019)