A Smart Health Assistant via DALI Logical Agents*

Stefania Costantini[1], Lorenzo De Lauretis[1], Claudio Ferri[1], Jessica Giancola[1], and Fabio Persia[1]

University of L'Aquila, Italy

Abstract. In this paper we describe an application of DALI logical agents in the eHealth field, where DALI is an agent-oriented language developed by our research group, based on the logic programming paradigm. Thanks to DALI, we have been able to build intelligent agents via very few lines of code, that implement significant reasoning tasks. In our project, we build a Smart Health Assistant, that is a Multi-Agent System (MAS) developed by using the DALI language in conjunction with Redis. This Smart Health Assistant is able to cooperate with an app situated on the patient's mobile phone, to capture health data from sensors, elaborate it, and then send the current health status to the patient and possibly to the doctor, thus providing real-time health monitoring.

Keywords: Artificial Intelligence · DALI · eHealth · Logic Programming

CEUR-WS.org/Vol-3002/paper15.pdf

1 Introduction

Artificial Intelligence is a field of computer science aimed at understanding and building intelligent entities, dating back to the period after the World War II, and whose developments have increased enormously in recent years. It is a truly universal field, as it can be applied in any intellectual context, from games such as chess to many areas of medicine. In the last century the first intelligent agents were born, but what is an agent? As in [29]: "An agent is anything that can be seen as a system that perceives its environment through sensors and acts on it through actuators". That is, an agent is composed of one or more sensors, which may be cameras, infrared sensors, ultrasonic sensors and so on, which provide a perception of the external world, and actuators, which may be, for example, motors, which allow the agent to perform actions.

However, what can make agents 'intelligent' is their software 'core', which processes the input received and obtains the best response. Nowadays, there are many languages that can be used for programming Artificial Intelligence, including Python, C++ and Prolog. There are many different implementations of the latter, among which in particular SICStus Prolog [10], which is the basis

* Copyright © 2021 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

of the DALI [11, 21] agent-oriented language. DALI is in fact a logical language; so, similarly to Prolog, it is based on the logic programming paradigm.

The project that we present in this paper focuses on the field of healthcare, with particular attention to machine-supported assistance of persons with special needs – in particular, the ill, the elderly and the disabled – to improve their independence and general quality of life. The project has been developed in cooperation with the ESH Center of Excellence for Hypertension of the University of LAquila, directed by Prof. Claudio Ferri. The provision of quality healthcare in a cost-effective way is in fact a critical issue in all countries, due to the ageing population, the reappearance of diseases that were considered extinct, and the development of challenging new issues, such as the Ebola and Covid-19 pandemics. Intelligent healthcare systems can be helpful to cope with such issues in the interest of patients, doctors, personnel and all other parties involved, including patients families. Hardware devices are gradually becoming less expensive, including wearable devices for detecting patients data (such as, e.g, blood pressure, heart beating, fever, etc). In fact, the prototype system that we implemented exploits some cost-affordable wearable devices. The project must be seen as the first step towards an overall system that we intend to devise, that will be demonstrably capable to ensure high-quality and sustainable health services while addressing ethical challenges, since ethical behaviour must be in our view an inherent feature of such a system. This kind of system may constitute in perspective a long-term solution in the everyday management of an ageing population, even in dramatic and pressing circumstances, such as a pandemic.

In particular, here we introduce a Smart Health Assistant, built upon the DALI language, using Multi Agent Systems (MAS), and in particular logical agents, as the basic technique. The main agent of our MAS is "SmartHealthHelper", that will be described in detail below, and constitutes a patient's "Personal Assistant Agent". This simple system is intended as a first step towards a much more comprehensive system, on the line of the proposal exposed in [1].

The paper is organized as follows. In Section 2, we introduce the DALI language. In Section 3, we describe the communication techniques available in DALI. In Section 4 we show the agents composing the Smart Health Assistant MAS, whereas in Section 5 we illustrate the overall system architecture. Eventually, in Section 7 we highlight final remarks concerning possible future scenarios.

2 DALI

The material exposed in this section and in the next one reports previous work on logical agents to provide the notions needed in the subsequent sections. This material is largely taken (in some parts literally, to be faithful to well-established terminology) from [11, 21, 23, 22, 20, 12, 13, 17, 18] and from the DALI web site[1].

DALI is a logical language derived from Prolog, and, similarly to it, is based on the logic programming paradigm. DALI has been fully implemented, on the

[1] https://github.com/AAAI-DISIM-UnivAQ/DALI

basis of a fully logical semantics [22]. Logic programming, unlike object-oriented programming which is based on solving a problem (i.e. traditional Java, Python, C++ etc.), focuses on describing the logical structure of the problem by defining clauses (also called 'rules'); specifically, logic programming adopts the Horn's fragment of clause logic [31]. In particular, DALI provides a number of mechanisms that enhance the basic clause language, in order to support the Agent-Oriented paradigm. This language offers the possibility to implement one or more intelligent agents leaving architectural freedom to the designer, i.e., without having to rely on a predefined architecture to define the agents.

To do so, DALI agents can employ constructs, such as events, goals and actions, distinguishable by post-fixed notation, to perform their task. DALI agents are in particular composed of a set of rules based on the concepts of *Reactivity*, *Proactivity*, *Sociality*, and *Memory*.

Besides DALI, other computational-logic-based agent-oriented languages and frameworks to specify agents and Multi-Agent Systems (MAS) have in fact been defined over time; for a survey of these languages and architectures, each one with its distinguished features, the reader may refer, among many, to [3, 26, 7].

3 Communication in DALI and Ontologies

The DALI communication architecture [23] implements the DALI/FIPA protocol, which consists of the main FIPA primitives[2], plus few new primitives which are peculiar to DALI. Primitives that can be used in DALI include:

- *send_message(External_Event, Sender_agent)*; its purpose is to trigger the reactive rule related to the event in the argument, in the agent receiving the message.
- *propose(Action, [Condition1,.., ConditionN], Sender_agent)*; with this primitive the sending agent invites the receiving agent to perform the action in the argument. In order to be able to accept, the receiver must first check the conditions specified in the list as the second parameter.
- *accept_proposal(Accepted_Action, [Condition1,..., ConditionN], Sender_agent)*; the agent who received the proposal, after checking the conditions, accepts. It is closely related to the previous primitive *propose*.
- *reject_proposal(Rejected_action, [Reason1,..., ReasonN], Agent_Sender)*; the agent who received the proposal rejects it, producing the list the unsatisfied conditions. It is closely related to the previous primitive *propose*.
- *failure(Action_failed, motivation(Reason), Sender_agent)*; the agent who received the proposal explains to the sending agent that the action was attempted but for some reason failed. This often happens because, although the conditions have been verified, the recipient agent's logic program does not allow the action to be performed. It is closely related to the previous *propose* primitive.

[2] FIPA is a widely used standardized ACL (Agent Communication Language), cf. http://www.fipa.org/specs/fipa00037/SC00037J.html for language specification, syntax and semantics.

– *cancel(Action_to_delete, Sending_agent)*; the agent requests to cancel the request for an action isued by another agent. It is closely related to the previous *propose* primitive.

These primitives are to be used within the distinguished action
messageA(Agent_receiver, primitive(..)).

Each agent has its own customizable filter for incoming and outgoing messages, composed by user-definable metarules which are to be specified in a file called "communication_agent_name.con". In particular, outgoing messages are filtered through the rule:

tell(Recipient, Sender, Primary(..)) :- Condition1,.., ConditionN.

The message will then be sent if the rule for the primitive used is present in the communication file, and if the conditions are met. It is also possible not to enter conditions, but to use 'true' instead, which implies that the message will always be sent. Incoming messages are filtered by the rule:

told(Sender, Primitive(..)) :- Condition1,.., ConditionN.

The message will therefore be received if the rule for the primitive used is present in the file, and the conditions are checked (or there is 'true'). In addition, there are rules for meta-reasoning which allow the agent to consult its knowledge and ontologies for understanding incoming messages. Ontologies, in the computer science field, are a very powerful tool to describe the world that surrounds us; they provide formal models of domain knowledge that can be exploited by intelligent agents [25]. An ontology consists of a set of rules and meta-rules that define terms and relationships between terms, including user-defined relationships as well as general ones, such as equivalence, generalization or inverse. Thus, an Ontology is a knowledge base that an agent can use, and that can be expressed in a tractable way in logical terms, see., e.g., [8, 2]. DALI agents can in fact make use of Ontologies written in tractable subsets of OWL (Ontology Web Language).

This part of the DALI language has been necessary to establish significant communication patterns among the agents composing our system. The importance of the *propose* primitive and related ones is that when a doctor proposes a medicine or a treatment to the patient's personal agent, such proposal should be either accepted or, if declined, reasons for that should be made explicit.

In Figure 5, we can see the architecture of our system, including the communication among agents. For the communication between DALI agents, we used FIPA ACL messages, that are part of the FIPA standard. To communicate with the others components of our system, such as 'Publisher.py', 'Subscriber.py' and the database, instead, we used both REDIS and Http GET requests, because, in this way, it is more efficient and faster.

4 Smart Health Assistant Overview

The goal of this project is to implement an intelligent system which, by analyzing data, is able to recognize the patient's state of well-being or discomfort in both

short and long term, and which is therefore able to assess the seriousness of the situation and, if necessary, alert a human doctor or even call the Emergency Service.

The main agent of the system is the "SmartHealthHelper" agent, which can be seen as the patient's Personal Assistant agent, which helps the patient in the supervision of her health status. However, our aim is that the agent will also be able to assist the patient in her daily life; for instance, it can do that by reminding her not only to take the prescribed medications but also about other tasks to accomplish, and aiding her in performing everyday activities.

This system is written in DALI, plus other auxiliary sub-modules. Specifically, thanks to an interface implemented in Python 3, our agent can collect from a database the vital parameters recorded by an app installed on the patient's mobile phone. Such data are partly collected automatically by medical devices, and partly entered autonomously by the patient upon request from the app.

Using Redis [9] (which is an archive of data structures used as a database, cache and, as in this case, as a message broker), the app can send the obtained parameters to the MAS that we have realized, in particular to the DALI "SmartHealthHelper" agent, which analyzes them in order to provide immediate feedback. Moreover, such parameters are stored (with timestamps) in internal lists, so as to keep track of the trend and to check for any abnormal situations via complex event processing [19, 15, 30, 28, 27] applicable to the medical [32] and robotic [24] domains. This is carried out also by taking into account previous illnesses and the feedback given by the patient via the app on his/her mental and physical state.

In the MAS there is also a "doctorAgent", that is consulted if the smartHealthHelper agent detects situations that need to be evaluated. The "doctorAgent" is supposed to interact if necessary with a human doctor, so as to provide "SmartHealthHelper" with the doctor' advice. The doctor agent's feedback is returned to Python via Redis, and is sent to the interface that interacts with the database.

In particular, the system performs the following tasks:

- *The Python module*: Collects patient data from the database and sends it to the MAS, and then processes the feedback received from the MAS and sends it to the database
- *The SmartHealthHelper agent*:
  - Analyses the patient's vital parameters (heart rate, saturation, minimum and maximum pressure, temperature, weight) taken by wearable devices or by the patient herself, returns immediate feedback on each parameter and stores them in the corresponding list.
  - Every time new parameters are entered, it compares the last three values stored in the list, and detects a dangerous situation (such as tachycardia/bradycardia in case of heart rate, hypoxia in case of saturation, and hypertension/hypotension in case of pressure), if all the three values are below or exceed a given threshold. The number of last measurements to be investigated (i.e., *three*, in this preliminary case study) is suggested

by medical doctors. In the next versions of the system, we will apply more sophisticated techniques based on temporal windows, which will specifically depend on the analyzed parameter.
    - Analyzes the symptoms that the patient reports.
    - In case of a dangerous situation, it contacts the doctorAgent which will provide feedback on what to do.
    - For each therapy entered, i.e., a medicine to be taken every day, it sends a reminder to the user at the right time;
    - Periodically, it sends a reminder for temperature, blood pressure and weight measurements.
    - With every new entry of a weight value, it compares it with the previous one and returns a feedback indicating if the user has lost or taken weight, how many kilos, and also communicates it to the doctor agent.
- *DoctorAgent*:
    - Receives messages from the *smartHealthHelper* agent, analyzes the communicated problem, e.g., fever or tachycardia, and responds by suggesting a medication to take or, in case of serious danger, it suggests to go to the Emergency Room. It also provides feedback on when to carry out the next measurement. Such feedback can partly be provided automatically, and partly by asking a human doctor.

4.1 Testbed

During the development of the project, in order to test the system, we used the (anonymous) data voluntarily provided by a female patient aged over 80, suffering from severe heart failures and occasional angina, with comorbidities [3]. In particular, we represented:

- the therapies (which medications at which time of the day);
- symptoms that can be treated by readjusting the therapies;
- recurring expected symptoms that require additional specific medications to be taken;
- situations of danger/alarm that require immediate intervention of a medical doctor, or transportation to the hospital.

4.2 Internal Event Example

In the code shown in Fig. 1, which refers to the *smartHealthHelper* agent, the vital parameters are analyzed over the long term.

In particular, for each parameter, the system checks whether the last three values are above or below a limit value, and if so, it asserts the relevant problem. In addition, it checks whether the patient is experiencing arm or chest pain (symptoms of angina), and if so, a situation of potential danger is asserted. If a dangerous situation is asserted, then the system will alert the *DoctorAgent* agent, that will be able to do a diagnosis and to propose countermeasures, based on the data obtained by sensors, and possibly via the advice provided by a human doctor.

[3] after testing the system, she gave us positive feedback

```
angina(X) :- trouble(X), X == 'dol_petto', retract(trouble(X)); trouble(X), X ==
'dol_braccia', retract(trouble(X)).

angina_ev(X) :- after_evp_time(sensazione(_),0,0,0,1), angina(X).
angina_evI(X) :> assert(danger(X)), retract(angina(X)).

whats_up :- after_evp_time(angina_ev(_),0,0,10,0).
whats_upI :> Message = 'how', mas_send(Message).

tachicardia_ev :- after_evp_time(battito(_),0,0,0,1),lists(heartbeat,LH),
last_values_mag(LH,120,Res), Res == 'si'.
tachicardia_evI :> assert(danger(tachicardia)).

bradicardia_ev :- after_evp_time(battito(_),0,0,0,1),lists(heartbeat,LH),
last_values_min(LH,60,Res), Res == 'si'.
bradicardia evI :> assert(danger(bradicardia)).

ipossia_grave_ev :-  after_evp_time(saturazione(_),0,0,0,1), lists(saturation,LS),
last_values_min(LS,90,Res), Res == 'si'.
ipossia_grave_evI :> assert(danger(ipossia_grave)).

presmin_alta_ev :-  after_evp_time(presminima(_),0,0,0,1),
       lists(diastolic_pressure,LMin), last_values_mag(LMin,90,Res),
       (Res == si -> print('Pressione minima alta'),nl, true;
       retract(presmin_alta), false).
presmin_alta_evI :> assert(presmin_alta).

presmin_bassa_ev :-  after_evp_time(presminima(_),0,0,0,1),
       lists(diastolic_pressure,LMin), last_values_min(LMin,60,Res),
       (Res == si -> print('Pressione minima bassa '),nl, true;
       retract(presmin_bassa), false).
presmin_bassa_evI :> assert(presmin_bassa).

presmax_alta_ev :-  after_evp_time(presmassima(_),0,0,0,1),
       lists(systolic_pressure,LMax), last_values_mag(LMax,140,Res),
       (Res == si -> print('Pressione massima alta'),nl, true;
       retract(presmax_alta), false).
presmax_alta_evI :> assert(presmax_alta).

presmax_bassa_ev :-  after_evp_time(presmassima(_),0,0,0,1),
       lists(systolic_pressure,LMax), last_values_min(LMax,90,Res),
       (Res == si -> print('Pressione massima bassa '),nl, true;
       retract(presmax_bassa), false).
presmax_bassa_evI :> assert(presmax_bassa).
```

Fig. 1. A sample of the code used to detect possible health problems

4.3 Call the doctor!

The *smartHealthHelper* agent will resort to the doctorAgent in case a dangerous situation is detected. In Fig. 3, the is the corresponding piece of code. The doctorAgent will return a piece of advise, that will be then displayed to the patient via the app.

The code reported in Fig. 2 shows how the *smartHealthHelper* agent detects alarming situations, either by comparing different measures of, e.g., pressure, over time, or by being informed by the patient herself about alarming symptoms.

```
ipotensione_ev :- after_evp_time(presmax_bassa_ev,0,0,0,2), presmin_bassa,

    presmax_bassa.
ipotensione_evI :> assert(danger(ipotensione)), retract(presmin_bassa),

    retract(presmax_bassa).

ipertensione_ev :- after_evp_time(presmax_alta_ev,0,0,0,2), presmax_alta,

    presmin_alta.
ipertensione_evI :> assert(danger(ipertensione)), retract(presmax_alta),

    retract(presmin_alta).

i_have_fever(X) :- after_evp_time(temperatura(_),0,0,0,1), fever(X,Y), Y == 'high'.
i_have_feverI(X) :> messageA(doctorAgent, send_message(febbre(X),Ag)),

    retractall(fever(_,_)),nl, print('Il paziente ha la febbre a '), print(X).
```

Fig. 2. A sample of the code used to detect possible alarming health problems

In both cases (Fig. 3), a message is sent to the *doctorAgent*, and its suggestion is displayed to the patient.

```
call_doctor(X) :- danger(X).
call_doctorI(X) :> messageA(doctorAgent, send_message(malore(X),Ag)),

    print('Il paziente ha '), print(X), retract(danger(X)).

doctor_suggestsE(X) :> const_separator(S),print('Il dottore suggerisce: '),

    print(X),nl,
    atom_concat('doctor',S,Mes), atom_concat(Mes,X,Message), mas_send(Message).
```

Fig. 3. A sample of the code used to communicate with the *doctorAgent*

4.4 doctorAgent code excerpt

In Fig. 4 we show part of the code concerning the doctorAgent. In particular, the doctor in this sample piece of code is able to provide advice to the patient, for instance to drink less in case of a sudden gain of weight (that in a patient suffering from heart failure - which is the case of the monitored female patient aged over 80 - implies a fluid retention which can be very dangerous for the heart). The agent is also able to provide advice for medicines to take, e.g., paracetamol for fever. In case everything fails, it suggests going to the Emergency Room. In perspective and with the advice of the medical doctors with whom we cooperate, the range of reasoning and advice capabilities of this agent will be expanded.

```
variazione_pesoE(X) :> if(X > 2, assert(feedpeso('drink_less')), true).

maloreE(X) :> print('Paziente suffers from '), print(X), assert(illness(X)).

febbreE(X) :> print('Patient has fever at '), print(X),
       if(X < 41, assert(illness('febbre')), assert(fever(X))).

take(medicine('Tachipirina')) :- illness(X), X == 'fever',
retract(illness(X)).
take(medicine('Propafenone')) :- illness(X), X == 'tachycardia',
retract(illness(X)).
take(medicine('sub-lingual_pill')) :- illness(X), X == 'chest_pain',
retract(illness(X));
           illness(X), X == 'dol_braccia', retract(illness(X)).

go_to_first_aid :- illness(X), X == 'severe_hypoxia', retract(illness(X));
           illness(X), X == 'bradycardia', retract(illness(X));
           illness(X), X == 'ipotension', retract(illness(X));
           illness(X), X == 'ipertension', retract(illness(X));
           fever(X), retract(fever(X)).
go_to_first_aidI :> print(' I suggest to go to the Emergency Room'),

messageA(smartHealthHelper,send_message(doctor_suggests('emergencyroom'),Ag))
.

suggestion(X) :- take(medicine(X)); feedpeso(X).
suggestionI(X) :> print(' suggest '), print(X),
  messageA(smartHealthHelper,send_message(doctor_suggests(X),Ag)),
  retractall(feedpeso(_)), retract(take(medicine(X))).
```

Fig. 4. A sample of the code used to send advice to the patient

5 System Architecture

In Fig. 6, the entire architecture of our system is depicted. As mentioned before, the patient interacts with an app on her mobile phone, which then records the data on a MySQL database. The system, through Http 'GET' calls, retrieves the data from the database and sends them to the MAS using the "message broker" functionality of Redis, introduced in the previous section. Those data will be later processed by the MAS, which sends the appropriate feedback to the

database. So, the app will then inform the patient through notifications on the mobile phone, with the indication of the countermeasures to be taken.

Notice that the MAS receives initial information from the app about the medical history and prescribed therapies of a patient via a piece of Python code, that for the sake of completeness we report below only for the interested reader (Fig. 5).

6 Code Overview

In Figures 1 to 5, we can see pieces of the code we used to develop our system. In Figure 1, we can see a sample of the code, written in DALI, that is used to describe various events that can occur while monitoring. For example, if saturation value is less than 90, the event "ipossia_grave_ev" is triggered. This particular event means that the patient does not saturates well, thus does not have enough oxygen, and a danger event is asserted. Other codes in this Figure does the same, asserts something if a particular event is detected. The same logic can be applied in Figure 2. In Figure 3, we can see a sample of the code that is used to communicate with the doctorAgent. In case a danger event is detected, the system proceed to sending a message to the doctor. Then, if the doctor has some suggestion, it can even answer to the patient, giving an hint about the right thing to be done. In Figure 4, we can see a sample of the code that is used to automatically send advices to the patient. If, for example, bradycardia is detected, the system gives you the suggestion to go to a first aid center, because the situation may be dangerous. In Figure 5, we can see a semple of the code that is used to get the patient parameters and current therapies. The system is linked with a database, linked with a smart sensor array that monitors a number of parameters of the patient, such as heart rate, saturation, and so on. Using this particolar piece of code, the system can interact with the database, obtaining real-time information on the patient status.

7 Conclusions and future work

In its initial form, the DALI language was a very good language for programming intelligent agents, but it was not able to work in a context that was too complex, full of events to manage, and of external systems to interact with. Nowadays, instead, thanks to improvements over the years, DALI offers great possibilities of managing complex contexts with a lot of occurring events [11]. By exploiting a simple syntax with which the structure of a problem can be implemented, DALI allows an agent to draw conclusions efficiently and quickly and at the same time manage numerous internal and external events.

The Smart Health Assistant project, built on DALI, offers high-level logical reasoning capabilities, using very low resources, being able to run even on very low-specs machines. It receives a lot of raw data from the patient (such as heart rate, blood pressure, temperature, and so on..), and it is able to process such data very quickly, allowing the system to do real-time monitoring on the patient. If a

```
def getParameters(parameters, redisInstance, url_base):
  for parameter in parameters:
     request_url = url_base + "&" + parameter[1] + "=1"
     response = req.get(request_url).text
     if parameter[0] == 'stato' or parameter[0] == 'sensazione':
       array = response.lower().split(LIST_SEPARATOR)
       for feel in array:
         if feel:
           prologParam = parameter[0] + "(" + feel + ")"
           prolog = makeAtomic(prologParam)
           redisInstance.publish(CH_OUT, prolog)  # Pubblico sul canale il
           parametro raccolto

         else:
           prologParam = parameter[0] + "(" + response + ")"
           prolog = makeAtomic(prologParam)
           redisInstance.publish(CH_OUT, prolog)  # Pubblico sul canale il
           parametro raccolto

def run(patientName):
  R = redis.Redis()  # Istanza 1 di Redis
  url_base = "http://www.lorenzodelauretis.it/tesi/index.php?nome="  +
  patientName

  parameters = [["anni", "get_eta"], ["altezza", "get_altezza"], ["peso",
  "get_peso"], ["temperatura",  "get_temperatura"], ["saturazione",
  "get_saturazione"], ["battito",  "get_battito"], ["presminima",  "get_presminima"],
  ["presmassima", "get_presmassima"], ["stato", "get_stato"],
  ["sensazione","get_sensazione"]]
```

```
# Insert previous illnesses

request_url = url_base + "&get_patologia=1"
patologies = req.get(request_url)
patologiesList = patologies.text.split(LIST_SEPARATOR)

for patology in patologiesList:
   if patology:
      prolog = "patologia('" + patology.lower() + "')"
      prologAtomic = makeAtomic(prolog)
      R.publish(CH_OUT, prologAtomic)

# Insert prescribed therapies

request_url = url_base + "&get_terapia=1"
therapies = req.get(request_url).text
therapiesList = json.loads(therapies)

for therapy in therapiesList:
   orario = therapy['orario'].split(":")

   if therapy:
      prolog = "'" + therapy['medicina'].lower() + "','" +
      therapy['quantita'].lower() + "'," + orario[0] + "," + orario[1]
      prologAtomic = makeAtomic(prolog,  True)
      prologAtomic = "terapia(" + prologAtomic + ")"
      R.publish(CH_OUT, prologAtomic)
```

Fig. 5. Input patient's parameters and therapies

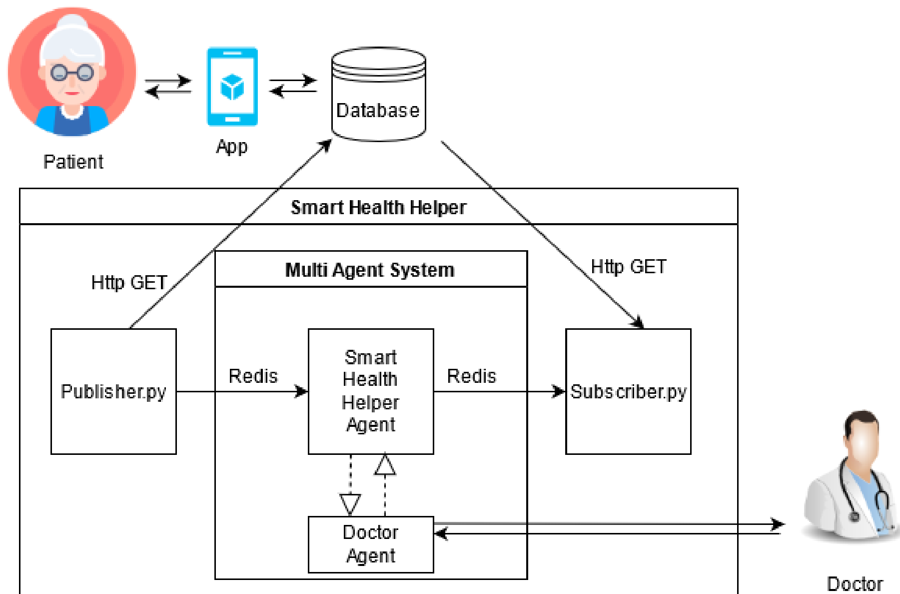

Fig. 6. Overall system architecture

critical situation is detected, the doctor (or even the first aid services) is alerted and the user receives notifications on what he should do to prevent injuries and stay healthy.

Currently, the system works well for basic situations; for instance, to check if the patient has fever or low pressure. In presence of several interacting medical conditions, the system is able to understand that "something bad" is happening, and consequently alert the doctor, but is still not able to do a precise diagnosis by itself.

In the future, we would like to implement forms of complex reasoning concerning complicated medical conditions (i.e. Pneumonia, Heart Attack, and so on...). Also, we intend to provide our application with access to medical information systems (e.g., patient databases, medical archives). The aim is to provide a better aid to the patients, decreasing the time that the system needs to contact a human doctor for a more precise diagnosis, and improving the information provided to the doctor, so as to effectively and timely help the patient.

Also, we intend to enrich the MAS with other agents, for instance agents representing specialist doctors that the patients has consulted or might consult, and institutions such as Heath Centers where the patients can perform various kinds of medical tests, and even Emergency Care and Hospitals. In perspective, the MAS would evolve at least in principle into a Multi-Context System (MCS), which is able to encompass not only agents but also kinds of "passive" entities, which can be queried to obtain results (e.g., medical knowledge bases or services for booking medical tests). An advantage of MCSs is the possibility of defining

standard communication modalities which such entities, via so-called "bridge rules". MCS's definition [4–6] was initially a bit restricted, but has been suitably extended in [16, 14], so that MCSs can now encompass logic-based agent, and data and ontologies exchange. The ultimate objective is to create an integrated system where the usage of available resources in a National or local health management system can be suitably planned, to the advantage of all parties involved.

Finally, we may notice that robotic hardware solutions are becoming increasingly effective, and so many inexpensive but flexibly usable robots are now available on the market. As a next step, we intend to install our main agent on a robot, to be situated at the patient's home so that, as said before, it can support the user also in her everyday tasks, detecting via an active observation of the user's activities the kind of help that the user might need at any moment.

Final note: we apologize for the many self-citations, due to the fact that this paper constitutes the last step of a long line of work.

References

1. Aielli, F., Ancona, D., Caianiello, P., Costantini, S., De Gasperis, G., Marco, A.D., Ferrando, A., Mascardi, V.: FRIENDLY & KIND with your health: Human-friendly knowledge-intensive dynamic systems for the e-health domain. In: Bajo, J., Escalona, M.J., Giroux, S., Hoffa-Dabrowska, P., Julián, V., Novais, P., Pi, N.S., Unland, R., Silveira, R.A. (eds.) Highlights of Practical Applications of Scalable Multi-Agent Systems. The PAAMS Collection - International Workshops of PAAMS 2016, Proceedings. Communications in Computer and Information Science, vol. 616, pp. 15–26. Springer (2016)
2. Artale, A., Calvanese, D., Kontchakov, R., Zakharyaschev, M.: The dl-lite family and relations. J. Artif. Intell. Res. **36**, 1–69 (2009)
3. Bordini, R.H., Braubach, L., Dastani, M., Seghrouchni, A.E.F., Gómez-Sanz, J.J., Leite, J., O'Hare, G.M.P., Pokahr, A., Ricci, A.: A survey of programming languages and platforms for multi-agent systems. Informatica (Slovenia) **30**(1), 33–44 (2006)
4. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: Proc. of the 22nd AAAI Conf. on Artificial Intelligence. pp. 385–390. AAAI Press (2007)
5. Brewka, G., Eiter, T., Fink, M.: Nonmonotonic multi-context systems: A flexible approach for integrating heterogeneous knowledge sources. In: Balduccini, M., Son, T.C. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 6565, pp. 233–258. Springer (2011)
6. Brewka, G., Ellmauthaler, S., Pührer, J.: Multi-context systems for reactive reasoning in dynamic environments. In: Schaub, T. (ed.) ECAI 2014, Proc. of the 21st European Conf. on Artificial Intelligence. pp. 159–164. IJCAI/AAAI (2014)
7. Calegari, R., Ciatto, G., Mascardi, V., Omicini, A.: Logic-based technologies for multi-agent systems: a systematic literature review. Auton. Agents Multi Agent Syst. **35**(1), 1 (2021)

8. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Dl-lite: Tractable description logics for ontologies. In: Veloso, M.M., Kambhampati, S. (eds.) Proceedings, The Twentieth National Conf. on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conf. pp. 602–607. AAAI Press / The MIT Press (2005)
9. Carlson, J.: Redis in action. Simon and Schuster (2013)
10. Carlsson, M., Widen, J., Andersson, J., Andersson, S., Boortz, K., Nilsson, H., Sjöland, T.: SICStus Prolog user's manual, vol. 3(1). Swedish Institute of Computer Science, Kista, Sweden (1988)
11. Costantini, S., Tocchio, A.: A logic programming language for multi-agent systems. In: Logics in Artificial Intelligence, Proc. of the 8th Europ. Conf.,JELIA 2002. pp. 1–13. LNAI 2424, Springer-Verlag, Berlin (2002)
12. Costantini, S., Tocchio, A.: DALI: An architecture for intelligent logical agents. In: Proceedings of the Int. Workshop on Architectures for Intelligent Theory-Based Agents (AITA08). AAAI Spring Symposium Series (2008)
13. Costantini, S.: Self-checking logical agents. In: Osorio, M., Zepeda, C., Olmos, I., Carballido, J.L., Ramírez, R.C.M. (eds.) Proceedings of the Eighth Latin American Workshop on Logic, Languages, Algorithms and New Methods of Reasoning LA-NMR 2012. CEUR Workshop Proceedings, vol. 911, pp. 3–30. CEUR-WS.org (2012), invited Paper, Extended Abstract in Proceedings of AAMAS 2013, 12th Intl. Conf. on Autonomous Agents and Multi-Agent Systems
14. Costantini, S.: Knowledge acquisition via non-monotonic reasoning in distributed heterogeneous environments. In: Truszczyński, M., Ianni, G., Calimeri, F. (eds.) 13th Int. Conf. on Logic Programming and Nonmonotonic Reasoning LPNMR 2013. Proc. Lecture Notes in Computer Science, vol. 9345, pp. 228–241. Springer (2015)
15. Costantini, S., De Gasperis, G.: Complex reactivity with preferences in rule-based agents. In: Bikakis, A., Giurca, A. (eds.) Rules on the Web: Research and Applications - 6th Intl. Symposium, RuleML 2012, Proc. Lecture Notes in Computer Science, vol. 7438, pp. 167–181. Springer (2012)
16. Costantini, S., De Gasperis, G.: Exchanging data and ontological definitions in multi-agent-contexts systems. In: Bassiliades, N., Fodor, P., Giurca, A., Gottlob, G., Kliegr, T., Nalepa, G.J., Palmirani, M., Paschke, A., Proctor, M., Roman, D., Sadri, F., Stojanovic, N. (eds.) Proceedings of the RuleML 2015 Challenge, the Special Track on Rule-based Recommender Systems for the Web of Data, held at RuleML 2015. CEUR Workshop Proceedings, vol. 1417. CEUR-WS.org (2015)
17. Costantini, S., De Gasperis, G., Nazzicone, G.: Exploration of unknown territory via DALI agents and ASP modules. In: Omatu, S., Malluhi, Q.M., Rodríguez-González, S., Bocewicz, G., Bucciarelli, E., Giulioni, G., Iqba, F. (eds.) Distributed Computing and Artificial Intelligence, 12th International Conference, DCAI 2015. Proceedings. Advances in Intelligent Systems and Computing, vol. 373, pp. 285–292. Springer (2015)
18. Costantini, S., De Gasperis, G., Nazzicone, G.: DALI for cognitive robotics: Principles and prototype implementation. In: Lierler, Y., Taha, W. (eds.) Practical Aspects of Declarative Languages - 19th International Symposium, PADL 2017, Proceedings. Lecture Notes in Computer Science, vol. 10137, pp. 152–162. Springer (2017)
19. Costantini, S., De Gasperis, G., Pitoni, V., Salutari, A.: DALI: A multi agent system framework for the web, cognitive robotic and complex event processing. In: Monica, D.D., Murano, A., Rubin, S., Sauro, L. (eds.) Joint Proceedings of

the 18th Italian Conference on Theoretical Computer Science and the 32nd Italian Conference on Computational Logic. CEUR Workshop Proceedings, vol. 1949, pp. 286–300. CEUR-WS.org (2017), http://ceur-ws.org/Vol-1949
20. Costantini, S., Dell'Acqua, P., Pereira, L.M.: A multi-layer framework for evolving and learning agents. In: M. T. Cox, A.R. (ed.) Proceedings of Metareasoning: Thinking about thinking workshop at AAAI 2008, Chicago, USA (2008)
21. Costantini, S., Tocchio, A.: The DALI logic programming agent-oriented language. In: Alferes, J.J., Leite, J.A. (eds.) Logics in Artificial Intelligence, 9th European Conference, JELIA 2004 2004, Proceedings. Lecture Notes in Computer Science, vol. 3229, pp. 685–688. Springer (2004)
22. Costantini, S., Tocchio, A.: About declarative semantics of logic-based agent languages. In: Baldoni, M., Endriss, U., Omicini, A., Torroni, P. (eds.) Declarative Agent Languages and Technologies III, Third International Workshop, DALT 2005, Selected and Revised Papers, Lecture Notes in Computer Science, vol. 3904, pp. 106–123. Springer (2005)
23. Costantini, S., Tocchio, A., Verticchio, A.: Communication and trust in the DALI logic programming agent-oriented language. Intelligenza Artificiale **2**(1), 39–46 (2005), Journal of the Italian Association AIxIA
24. D'Auria, D., Persia, F.: A collaborative robotic cyber physical system for surgery applications. In: 2017 IEEE International Conference on Information Reuse and Integration (IRI). pp. 79–83 (2017). https://doi.org/10.1109/IRI.2017.84
25. De Lauretis, L., Costantini, S., Letteri, I.: An ontology to improve the first aid service quality. In: 2019 IEEE International Conference on Systems, Man and Cybernetics (SMC). pp. 1479–1483. IEEE (2019)
26. Garro, A., Mühlhäuser, M., Tundis, A., Baldoni, M., Baroglio, C., Bergenti, F., Torroni, P.: Intelligent agents: Multi-agent systems. In: Ranganathan, S., Gribskov, M., Nakai, K., Schönbach, C. (eds.) Encyclopedia of Bioinformatics and Computational Biology - Volume 1, pp. 315–320. Elsevier (2019)
27. Helmer, S., Persia, F.: High-level surveillance event detection using an interval-based query language. In: 2016 IEEE Tenth International Conference on Semantic Computing (ICSC). pp. 39–46 (2016)
28. Helmer, S., Persia, F.: Iseql, an interval-based surveillance event query language. International Journal of Multimedia Data Engineering and Management (IJMDEM) **7**(4), 1–21 (2016)
29. Norvig, P., Russell, S.J.: Intelligenza artificiale. Un approccio moderno (2010)
30. Piatov, D., Helmer, S., Dignös, A., Persia, F.: Cache-efficient sweeping-based interval joins for extended allen relation predicates. VLDB J. **30**(3), 379–402 (2021). https://doi.org/10.1007/s00778-020-00650-5, https://doi.org/10.1007/s00778-020-00650-5
31. Rümmer, P., Hojjat, H., Kuncak, V.: Classifying and solving horn clauses for verification. In: Working Conference on Verified Software: Theories, Tools, and Experiments. pp. 1–21. Springer (2013)
32. Russo, R., DAuria, D., Ciccarelli, M., Della Rotonda, G., DElia, G., Siciliano, B.: Triangular block bridge method for surgical treatment of complex proximal humeral fractures: theoretical concept, surgical technique and clinical results. Injury **48**, S12–S19 (2017). https://doi.org/https://doi.org/10.1016/S0020-1383(17)30651-4, https://www.sciencedirect.com/science/article/pii/S0020138317306514, cIO Perspective: Various Focuses on Traumatology