A Python-based Assistant Agent able to Interact with Natural Language

Fabio Longo, Corrado Santoro
University of Catania
Department of Mathematics and Informatics
Viale Andrea Doria, 6
95125 - Catania, ITALY
EMail: flongo@policlinico.unict.it, santoro@dmi.unict.it

***Abstract*—This paper describes the software architecture and functionalities of an assistant agent, developed by the authors, able to interact with the user through the natural language. The agent is implemented by means of PROFETA, a Python-based BDI engine developed within the author's research group. The agent is composed of two parts: *(i)* the *speech-to-text* and *text-to-speech* services, and *(ii)* the *reasoning engine*. As for the former part, the agent exploits Clould services (in particular those provided by Microsoft Bing and IBM Watson); to this aim, a flexible software architecture is designed in order to connect first-class entities of PROFETA (i.e. sensors and actions) to the cloud world. The reasoning engine is instead designed by means of the declarative language provided by PROFETA: utterances said by the user become PROFETA beliefs that can, in turn, trigger reasoning rules. In order to show effectiveness of the solution, a case-study of speech-based interaction to browse Wikipedia is presented.**

I. Introduction

With the advent of smartphones, the technological advances in artificial intelligence made it possible the implementation of speech-based assistant agents, like Siri or the Google Assistant. Being predicted in many science fiction movies, speech assistants are kind of *personal agents* that represent the natural evolution of assistant agents introduced in the '90 in application software to help the user in her/his day-to-day activities [7], [18], [17], [6], [16]. But while such kind of assistants were strongly specialised for the activities proper of the associated application, the aim of speech assistants is more ambitious: to help the user in any kind of day-to-day activities. We indeed expect that such entities should be able to interpret—and execute the associated tasks—utterances like *"Play some jazz"*, *"Order coffee pods"*, *"Book me a flight to London"*, etc.

All of the exemplified activities however are not simple "commands", but, in general, could require a form of—more or less complex—interaction; as an example, at the first phrase above the assistant could answer something like *"I've found some tracks by Chet Baker, are they ok for you?"*, and the user could reply *"No, I prefer Thelonious Monk"*; placing an order could imply to know the price and delivery time of some offers, and the user (always by means of speed-based interaction) could be asked to make a selection. The concept is that we, as users, expect that speech-based assistants can establish a meaningful dialogue with us, even if we know that the peer is not a human being but an artificial system with obvious limited capabilities and intelligence.

The technological aspects behind such a form of assistance are multiple: not only a good Natural Language Processing (NLP) engine is needed, but also a *reasoner tool* is mandatory, that should be able to understand the context of discussion and interpret the meaning of sentences accordingly. The NLP engine has the task of performing speech-to-text processing and, having obtained the text part, applying a syntax analyser, in order to extract the meaningful parts of the phrase and classify the lemmas. Once the lemmas have been extracted, they have to be interpreted in order to catch the meaning and then execute the proper actions; this operation is usually performed by using tools that implement forms of—more or less flexible—string pattern matching. But pattern matching usually does not suffice for a powerful natural language interaction; indeed meaning of utterances is also strongly based on the evolution of *dialogues*, therefore what happened in the previous interactions of the dialogue is strongly important for a correct interpretation: in other words, pattern matching must be integrated with *state-* or *knowledge-data*. In this sense, what is needed is a tool that is able to support *logic-/knwoledge-based programming* and, among such kind of tools, the PROFETA [12], [13], [11], [10] programming platform appears an interesting solution since it is able to provide all the cited characteristics.

In this context, this paper presents the software architecture of a personal assistant agent based on the PROFETA programming platform and able to interact with a human by using the speech and the natural language. The solution is based on a software architecture that, by means of the integration of cloud services with PROFETA, is able to perform speech-to-text and text-to-speech operations. Interpreted phrases then become *beliefs* and can thus be used to trigger *production rules* that drive actions of the agent. As a case-study, the paper reports the application of the software architecture to the implementation of agent able to query Wikipedia on the basis of the questions provided by human user.

The paper is structured as follows. Section II provides an overview of PROFETA. Section III describes the architecture and functionalities of the speech-to-text/text-to-speech interface. Section IV presents the case-study. Section V concludes

the paper.

II. Overview of PROFETA

PROFETA[1] [12] is a Python platform for programming the behaviour of an autonomous system (agent or robot) based on the *Belief-Desire-Intention* paradigm [8], [15]. It allows a programmer to implement the behaviour of an agent by using two basic concepts: *beliefs* and *plans*.

Beliefs represent the data part of an agent program, i.e. the *knowledge* of the agent and are expressed by using logic predicates, i.e. atomic formulae with ground terms; a belief can be generated on the basis of data coming from the agent reference environment or as a result of a reasoning process.

Plans represent the computational part of an agent program and are expressed as *production rules*, triggered by a specific event which can be the assertion/retract of a belief or the request to achieve a specific goal; the result of a plan is a sequence of *actions* that represent what the agent has to do as a response to that event. Plans are written using Python statements that however are interpreted, by the PROFETA engine, as *declarative production rules*. The syntax is inspired by AgentSpeak [9]. A plan includes a *head* and a *body*; the *head* that is composed of the *triggering event* plus a *condition*, i.e. a predicate on the knowledge that must be met for the plan to be triggered; the *body* is the list of actions that must be executed when the plan is triggered.

The interaction of a PROFETA agent with the environment is performed by using two types of software entities. *Actions*, already cited in the context of plans, are responsible of acting onto the environment (as the name suggests), and are executed as a result of the triggering of a plan. *Sensors* are instead software entities with the task of polling or receiving data from the environment or other external entities; a sensor having gathered a useful information can generate a belief thus transforming data in agent knowledge; as a result, a belief generated by a sensor can enrich agent's knowledge and/or trigger a plan thus provoking the reaction of the agent.

PROFETA engine includes a scheduler that runs a loop continuously executing the following activities:

1) **Sensor Activation.** All defined sensors are activated and their polling code is executed; if a sensor generates a belief, it is processed and placed into an *event queue*.
2) **Event Handling.** An event is extracted from the event queue and all applicable plans are searched for.
3) **Plan Selection.** For each applicable plan, the condition is tested and, if true, the plan is selected for execution.
4) **Plan Execution.** The actions of the selected plan are executed sequentially.

For more details about PROFETA internals, working scheme, syntax and semantics, the reader can consult the relevant bibliography [12], [13], [11], [10].

[1] http://github.com/corradosantoro/profeta

III. The STT/TTS Interface

Implementing an agent able to interact with natural language implies to include and use speech-to-text (STT) and text-to-speech (TTS) services. Although many libraries exist (like Sphinx [1], for example), Cloud-based services appear a more interesting solution, mainly because they are continuously improved and upgraded, and made always more precise. Indeed, as for STT, we made some test using both the Sphinx library and the Microsoft Bing Speech service, and, as a result, we obtained a greater accuracy using the latter solution. On the other hand, for the TTS, we experienced the IBM Watson text-to-speech service [3], which gives a more natural feeling of human voice, than others Python-based engines like eSpeak [2], SAPI [4], etc.

In order to include STT and TTS services within PROFETA, the proper abstractions provided by the tool must be used; the STT is implemented as a PROFETA *sensor* that we called **hearer**, while TTS is encapsulated inside an *action* called **say**.

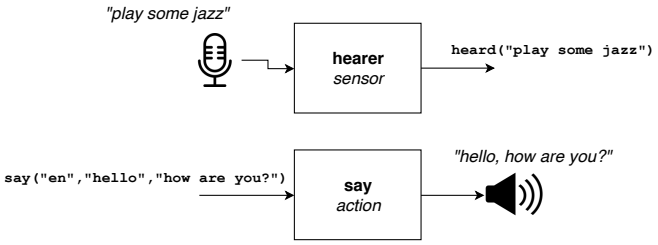

Fig. 1. The "hearer" sensor and "say" action

As Figure 1 show, the **hearer** has the task of sampling the microphone, sending audio data to the STT cloud service, and gathering the response as a string; as a consequence, a `heard()` belief is asserted whose term is the interpreted string. This assertion can, in turn, trigger the proper rules according to the agent program implemented.

The **say** action is used to let the agent pronounce a sentence (see Figure 1). The parameters of the action are strings: the first string is the language, while the other parameters represent the various parts of the sentence itself; action implementation concatenates such parts and sends the resulting string to the TTS cloud service; the reply will be the audio samples of the recited phrase that are sent to the audio device for playing.

From the implementation point of view, a sensor in PROFETA is simply made of class that extends the `Sensor` base class overriding the `sense()` method; this method must implement to code for data sampling, returning the relevant belief (or `None` if no data has been sampled). In a similar way, a PROFETA action is implemented as a sub-class of the `Action` base class and overriding the `execute()` method with the proper code. Given this, implementing the STT and TTS services appears a straightforward task, however some aspects must be taken into account. The first aspect regards the performances: the interaction with cloud services could introduce latencies that can affect the performances of

the overall system; indeed, in PROFETA, the invocation of `sense()` and `execute()` methods is made synchronously within the main interpretation loop of the agent program: if such methods experience delays (that are indeed unavoidable when a network transaction is performed), the overall performances are affected. The second aspect is related to the generality of the solution: even if, in our implementation, we decided to use Microsoft Bing for STT and IBM Watson for TTS, other engines are available in the web, therefore a software architecture able to let the programmer to easily change the desired engine is really opportune. Such aspects are dealt with in the next subsection.

A. The STT Interface

The STT interface is made by three basic classes and a specialised class; the relevant UML diagram is reported in Figure 2.

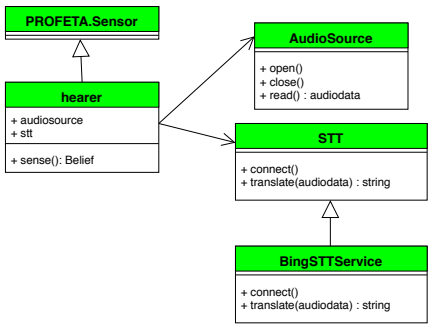

Fig. 2. The STT Interface

The principal entity is **hearer** which is a subclass of the basic PROFETA class *Sensor*. An instance of **hearer** contains a reference to other two objects: **AudioSource** and **STT**; the former has the objective of capturing audio data from the microphone while the latter implements the network transaction with a cloud STT services. **STT** is defined as an abstract class: its methods are empty and their implementation is left to a derived class that will include the code for the specific cloud service to be used. In our implementation, we subclassed **STT** as **BingSTT**, whose methods implement the interaction with the Microsoft Bing service.

The task of **hearer** is coded in its `sense()` method. First it invokes the `read()` method of **AudioSource** to retrieve audio data samples; this sampling is performed by entering in a loop which listens for incoming sounds from the microphone, filtering ambient noises properly, until any source of sound is perceived; then, the audio stream is caught until silence is identified again and the relevant data are returned as a result of `read()`. Subsequently, the **hearer** activates the **STT** by invoking the `translate()` method; this method receives the audio stream as input and is expected to return the translated string or `None` if translation is impossible. Such a result (if not `None`) is converted in lowercase[2] and the `heard()` belief is generated as a result value of the `sense()` method.

For the performance reasons already cited, the task described above must be executed in an asynchronous way with respect to the main PROFETA loop. This objective is achieved by encapsulating the **hearer** inside a **AsyncSensorProxy**, a library class provided by PROFETA which has the specific task of making a sensor asynchronous [14].

B. The TTS Interface

Text-to-speech is performed by a specific PROFETA *action*. Also in this case, it is preferable to have an asynchronous execution of the cloud interaction with respect to the main PROFETA loop. This is performed by exploiting the **AsyncAction** base class provided by the PROFETA library which is, in turn, derived from **Action**.

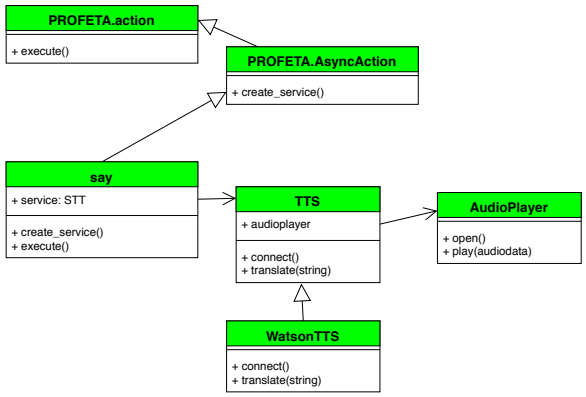

Fig. 3. The TTS Interface

The TTS interface, whose class diagram is reported in Figure 3, is composed of the following classes: **say**, **TTS**, **AudioPlayer** and **WatsonTTS**. Class **say** is the async-action which is directly invoked by the PROFETA program and coordinates all the text-to-speech activities. **TTS** is an abstract class that represents the cloud service and (in a way similar as to STT) must be subclassed with the implementation of the code for interaction with a specific TTS service; in our case, this is performed by the class **WatsonTTS** that includes the code for interaction with IBM Watson. Class **TTS** contains also a reference to **AudioPlayer**, which has the task of playing to the audio device the audio stream returned by the TTS service.

The working scheme of the TTS interface is based on a specific usage protocol that the programmer has to respect; in particular, in the **say** class, two methods must be overridden: `create_service()` and `execute()`. The former method is called when the class is instantiated and has the specific task of creating the **TTS** object and performing, in turn, the initial connection to the cloud service. The latter method is called when the action is explicitly invoked within a PROFETA program and contains the code that retrieves parameters, composes the string to say and invokes the `translate()` method of **TTS** that concretely executes text-to-speech and plays the resulting audio stream.

[2]This is required because, for syntax reasons, in PROFETA string constants must be in lowercase.

```
1  stage("main")
2  +start() >> [ +language("en"), show("starting...") ]
3  +heard("laura") >> [ random_greetings("GreetMessage"), say("en", "GreetMessage"), set_stage("wiki") ]
4
5  stage("wiki")
6  +heard("X") >> [ terms_to_belief("X") ]
7  +generic_phrase("change language") >> [ set_stage("language") ]
8  +generic_phrase("X") / language("L") >> [ wiki_search("L", "X", "Y"),
9                                                 say("L", " I have found the following options ", "Y") ]
10 +search("X") / language("L") >> [ wiki_say("L", "X") ]
11 +timeout() >> [ set_stage("main") ]
12
13 stage("language")
14 +start() >> [ say("en", "what language do you desire?") ]
15 +heard("english") >> [ +language("en"), say("en", " I've set english"), set_stage("wiki") ]
16 +heard("italian") >> [ +language("it"), say("en", " I've set italian"), set_stage("wiki") ]
17 +heard("french") >> [ +language("fr"), say("en", " I've set french"), set_stage("wiki") ]
18 +heard("X") >> [ say("it", " I've not understood the language that you desire or I'm not able to support it") ]
```

Fig. 4. The listing of Laura

IV. The Wikipedia Agent

The agent we developed as a proof-of-concepts for our STT/TTS interface is a simple assistant able to browse Wikipedia by means of natural language. We called the assistant *Laura* and this is the name the assistant itself respond to in its behaviour.

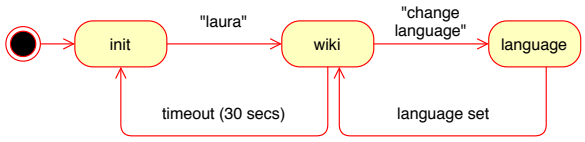

Fig. 5. The basic behaviour of Laura

Laura is implemented as a finite-state machine, sketched in Figure 5, in which each state (which is indeed a macro-state) represents a condition, in Laura's behaviour, corresponding to certain dialogue abilities. The basic working scheme is the following: in the initial state, **init**, the agent waits for her name in order to be "woken-up"; after that, Laura enters in the **wiki** state in which she is able to hear the term to be searched for in Wikipedia. In the **wiki** state Laura is able to respond to the following phrases:

- **"search *terms*"**, the specific terms are sent to wikipedia and, when the result page is returned, the summary data is recited using text-to-speech;
- **"change language"**, it makes Laura enter in the **language** state, letting the user to set a new language for both Wikipedia search and text-to-speech;
- **any other terms**, the list of possible options for the said terms is searched for in wikipedia and such a list is recited by Laura; if the user wants a specific term, s/he can ask it using the phrase **"search *terms*"**.

The **wiki** state is abandoned on the basis of two events: after a timeout of 30 of inactivity (in this case the state reached is once again **init**), or when the user says *"change language"*; in th latter case, Laura enters in the **language** state asking the user for the new language desired.

To support the cited activities, the following *beliefs* are used:

- `heard(terms)`. The belief already described in Section III used as output of the **hearer** sensor. It is defined as a *reactor* i.e. a belief that can only trigger PROFETA program plans but does not enters in the knowledge base[3].
- `search(terms)`. It is a reactor generated by a processing of data heard by the **hearer**: if the sentence said includes an explicit searching request, this reactor is asserted.
- `generic_phrase(terms)`. Like the previous one, it is a reactor generated by a processing of data heard by the **hearer**, but is generated if the sentence said **does not include** a specific searching request.
- `language(lang)`. It is a belief that stores, in the knowledge base, the settings relevant to the current language.
- `timeout()`. A reactor used to signal the 30 seconds of inactivity.

The complete PROFETA program that controls Laura's behaviour is reported in Figure 4 (the code of actions is not reported for brevity reasons, but their role is described in the text below).

The macro-states of the finite-state machine of Figure 5 are clearly identified since they are called *stages* in PROFETA and are used to specify that certain plans are valid (i.e. triggerable) in that state.

In stage *main*, the program waits for the assertion of `heard("laura")` reactor (this happens if the user pronounces "Laura") and, on the occurrence of such an event (line 3), it enters into the *wiki* stage. In such a reaction, a greeting message is recited: this message is generated by action **random_greetings()** that picks a random welcome string (from a predefined set) and bounds it to variable `GreetMessage`; the use of random greeting message is to avoid a repetitive behaviour, from Laura, that, in the long term, could boring the user.

In the *wiki* stage, the arrival of a `heard()` belief causes plan in line 6 to be triggered: the consequence is the call of action **terms_to_belief()** that has the basic task of interpreting

[3]See [12] for details about the kind of beliefs supported by PROFETA.

the command according to the cases listed above and depicted in Figure 5. If **"search *terms*"** is pronounced (e.g. "search Palermo"), the **terms_to_belief()** action asserts the **search()** reactor, using *terms* as parameters; this causes plan in line 10 to be executed: first the current language is retrieved, then action **wiki_say()** is executes which will search the terms inside Wikipedia reciting then the relevant summary text. If **"change language"** is pronounced, plan in line 7 is triggered and the agent enters into stage *language*, asking the user the new language to switch to; when the new language is successfully selected, the agent returns into the *wiki* stage (lines 15—17). If other terms are said, the plan in line 8 is executed that causes a generic search into Wikipedia for all the pages that are related to the terms themselves: the list of options is then recited by Laura.

V. Conclusions

This paper has described the software architecture and the working scheme of an assistant agent able to interact with the user with natural language. The basic aspects of the desired solution are the use of the PROFETA BDI tool as the execution platform and the organisation in a flexible software architecture in order to exploit cloud computing for speech-to-text and text-to-speech services.

The assistant implemented, called Laura, has the objective of helping the user in browsing Wikipedia with speech-based interaction. It served as a proof-of-concepts to understand the validity of the software architecture and the applicability of PROFETA to such kind of contexts.

Starting from such experience, we plan, in future work, to improve understanding abilities of Laura, including a library to parse natural language sentences (like NLTK [5]), also translating the parsed terms into a proper beliefs better representing the predicates of a common knowledge; the objective is to have an artificial system which can show a rational behaviour that can also be adopted in all contexts needing a form of specific user assistance.

References

[1] Cmusphinx: Open-source speech recognition toolkit. [Online]. Available: http://cmusphinx.github.io/

[2] espeak speech synthesizer. [Online]. Available: http://espeak.sourceforge.net/

[3] Microsoft speech application program interface. [Online]. Available: http://www.ibm.com/watson/services/text-to-speech/

[4] Microsoft speech application program interface. [Online]. Available: http://en.wikipedia.org/wiki/Microsoft_Speech_API

[5] Natural language toolkit. [Online]. Available: http://www.nltk.org/

[6] M. Bombara, D. Calì, and C. Santoro, "KORE: A multi-agent system to assist museum visitors," in *WOA 2003: Dagli Oggetti agli Agenti. 4th AI*IA/TABOO Joint Workshop "From Objects to Agents": Intelligent Systems and Pervasive Computing, 10-11 September 2003, Villasimius, CA, Italy*, 2003, pp. 175–178.

[7] J. M. Bradshaw, Ed., *Software Agents*. AAAI Press/The MIT Press, 1997.

[8] M. E. Bratman, *Intentions, Plans and Practical Reason*. Harvard University Press, 1987.

[9] M. d'Inverno and M. Luck, "Engineering agentspeak(l): A formal computational model," *Journal of Logic and Computation*, vol. 8, no. 3, pp. 233–260, 1998. [Online]. Available: http://eprints.ecs.soton.ac.uk/3846/

[10] L. Fichera, D. Marletta, V. Nicosia, and C. Santoro, "Flexible robot strategy design using belief-desire-intention model," in *Research and Education in Robotics - EUROBOT 2010 - International Conference, Rapperswil-Jona, Switzerland, May 27-30, 2010, Revised Selected Papers*, 2010, pp. 57–71.

[11] ——, "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.*, 2010.

[12] L. Fichera, F. Messina, G. Pappalardo, and C. Santoro, "A python framework for programming autonomous robots using a declarative approach," *Sci. Comput. Program.*, vol. 139, pp. 36–55, 2017.

[13] G. Fortino, W. Russo, and C. Santoro, "Translating statecharts-based into BDI agents: The DSC/PROFETA case," in *Multiagent System Technologies - 11th German Conference, MATES 2013, Koblenz, Germany, September 16-20, 2013. Proceedings*, 2013, pp. 264–277.

[14] F. Messina, G. Pappalardo, and C. Santoro, "Integrating cloud services in behaviour programming for autonomous robots," in *Algorithms and Architectures for Parallel Processing - 13th International Conference, ICA3PP 2013, Vietri sul Mare, Italy, December 18-20, 2013, Proceedings, Part II*, pp. 295–302.

[15] A. Rao and M. Georgeff, "BDI agents: From theory to practice," in *Proceedings of the first international conference on multi-agent systems (ICMAS-95)*. San Francisco, CA, 1995, pp. 312–319.

[16] A. D. Stefano, G. Pappalardo, C. Santoro, and E. Tramontana, "A multi-agent reflective architecture for user assistance and its application to e-commerce," in *Cooperative Information Agents VI, 6th International Workshop, CIA 2002, Madrid, Spain, September 18-20, 2002, Proceedings*, 2002, pp. 90–103.

[17] A. D. Stefano and C. Santoro, "Netchaser: Agent support for personal mobility," *IEEE Internet Computing*, vol. 4, no. 2, pp. 74–79, 2000.

[18] G. Weiss, Ed., *Multiagent Systems*. The MIT Press, April 1999.