Intelligent Environments 2021
E. Bashir and M. Luštrek (Eds.)

doi:10.3233/AISE210077

A Modular Architecture for Multi-Purpose Conversational System Development

Adrián ARTOLA [a,1], Zoraida CALLEJAS [a] and David GRIOL [a,2]
[a] *Dept. of Software Engineering, University of Granada, Granada, Spain*

Abstract. As the complexity of intelligent environments grows, there is a need for more sophisticated and flexible interfaces. Conversational systems constitute a very interesting alternative to ease the users' workload when interacting with such environments, as they can operate them in natural language. A number of commercial toolkits for their implementation have appeared recently. However, these are usually tailored to specific implementations of the processes involved for processing the user's utterance and generate the system response. In this paper, we present a modular architecture to develop conversational systems by means of a plug-and-play paradigm that allows the integration of developers' specific implementations and commercial utilities under different configurations that can be adapted to the specific requirements for each system.

Keywords. conversational systems, chatbots, modular architectures, natural language understanding, dialog management, conversational framework, human-machine interaction

1. Introduction

Intelligent Environments (IE) comprise a set of interconnected devices and sensors surrounding users to provide access to a plethora of information and services, which may create a great cognitive load in the users, specially in industrial settings (see e.g. [1]). Consequently, user empowerment can only be sustained in enhanced and more intuitive human-machine interactions.

Conversational systems have become very important to achieve this objective involving speech interaction and being able to process semantic and pragmatic knowledge [2, 3]. These interfaces have experienced a vast development in the recent years propelled by the widespread adoption of voice assistants and smart devices, the advances in Artificial Intelligence techniques and the increasing amount of data currently available to learn statistical models. These advances have created a whole new market for conversational systems, and in particular for IE assistants. The most renowned technological companies offer their language processing services both as assistants ready to be used

[1]A. Artola was with University of Granada during the realization of this work as part of his Master Thesis.
[2]Corresponding Author: David Griol, Periodista Daniel Saucedo Aranda SN, 18071 Granada, Spain; E-mail: dgriol@ugr.es.

(e.g. Amazon Alexa, Microsoft Cortana, Google Voice Assistant...) or as services that developers can employ to develop their own conversational systems or provide new skills to the already existing ones.

Such commercial tools offer different services, including natural language understanding and interaction management, which may vary in complexity. Task-oriented systems can be implemented easily with commercial toolkits such as DialogFlow or Amazon Lex. However, developers may find it difficult to combine the services from different vendors, specially natural language processing and dialogue management, as they are usually highly coupled in commercial systems.

Our aim is to offer a framework to develop multi-purpose conversational systems, which allows developers to use REST services to integrate their own implementations of specific modules of the system and also to combine them with the solutions provided by commercial toolkits.

The rest of the paper is organised as follows. Section 2 presents the state of the art about the different existing frameworks and architectures to develop conversational systems. Section 3 introduces our architecture as well as the terminology employed. Section 4 presents a practical implementation of our proposal, while Section 5 presents several configurations developed that show the appropriateness of the proposal to develop conversational systems in the same domain using different components from different vendors. Finally, Section 6 draws the conclusions and presents lines for future work.

2. Background

The typical pipeline to develop conversational interfaces consists of five components: automatic speech recognition, natural language understanding, dialogue management, natural language generation and text to speech synthesis [4].

Each component has specific purposes:

- The automatic speech recogniser receives the audio signal corresponding to the user's input and outputs a textual transcription. Typically, this module also provides confidence scores representing how confident the system is about the correctness of the returned text.
- The Natural Language Understanding module receives the text input and returns its semantic representation, as the perceived required task and the values for the necessary pieces of information required to perform it.
- The Dialogue Manager decides the next system action considering the semantic representation of the user's utterance, the previous dialogue history, the result of accessing the data repositories of the system, the specific regulations of the task, among others.
- The Natural Language Generator translates the action selected by the dialogue manager into one or more sentences in natural language (system prompt).
- Finally, the Text-to-Speech synthesizer translates the system prompt into an acoustic signal.

As mentioned before, there exist different frameworks for conversational systems development that try to accommodate some or all the previously described modules. McTear [3] presents a very complete and updated review of tools for developing dialogue systems, which are divided into tools for visual design, scripting tools, advanced toolkits and frameworks and research-based toolkits.

There is a huge variety of commercial toolkits offered by the largest IT companies. These toolkits offer all the necessary services to create a conversational system with high-level interaction and with improved possibilities to connect the developed system with different platforms, e.g., existing voice assistants, Telegram or Twitter. The most popular alternatives require the developer to define *intents* and *entities* for the understanding process, and dialogue management is determined according to the most relevant *intent* and the use of *slots* or active *contexts* that can be complemented with web services through *webhooks*.

Despite their flexibility, many times a considerable effort is required to handcraft a dialogue tree that is then coded into the system following the intents and entities format. Also these tools hinder the complexity and details of language and dialogue processing to developers, which is interesting to democratise conversational system development, but it may not be adequate for contexts in which developers want to have a broader control, including for example explainability of the decisions and security requirements, which are commonplace with IE interactions.

User confidence is key to IE [5]. As illustrated in [6], the capacity for self explainability is crucial for users to find IE trustworthy and reliable, specially when populated by smart assistants.

In academic settings, more sophisticated approaches are used for natural language understanding and dialogue management and can be used in controlled environments. This is the case of toolkits and implementation resources such as OpenDial [7], PyOpenDial [8], which also encompass a modular architecture, or the recent ConveRSE [9] and HRIChat [10], but are sometimes not straightforward to use in conjunction with commercial solutions.

Some of the previously mentioned alternatives do not offer the possibility of plugging different services for each of the modules that conform the conversational system. In fact, it is common to merge NLU and DM in commercial chatbot toolkits, as it is a trend to combine both options, specially when using machine learning to decide system responses.

However, in our approach, these two modules can be considered in isolation. This makes it possible to develop end-to-end systems in which both processes are performed at once, or to divide them into two independent services when it is necessary to have control over them. For example in hybrid systems where rules have to be applied into dialogue management. This is particularly interesting for IE as usually safety rules must be applied when operating the environments (e.g. always confirm when turning the oven on).

More versatile platforms like RASA [11] and DeepPavlov [12] provide more flexibility in the implementation. This paper presents exploratory work for a simple lightweight architecture that can be used to easily create conversational systems.

3. Proposed Architecture

We have chosen a service-oriented modular architecture based but not limited to the traditional pipeline. To foster interoperability, each part of the system is independent of the rest and the information shared is orchestrated by an *Information homogenisation module*. This new module receives the output of the Natural Language Understanding module and produces a technology-agnostic parsing into a *registry*, that can be subsequently translated into the format required by the rest of the services employed.

The dialogue history is stored as a board of *registries* that are categorised into *intents* (representing the intentions or actions required by the user) and *entities* (relevant pieces of information required to fulfil the task). The actions taken by the system are also incorporated into the board.

All the modules except the *Information Homogenisation (IH) Module* are divided into *infrastructure* and *superstructure*. On the one hand, the infrastructure is dedicated to connect all services and to interact with the container application. On the other hand, the superstructure is a specific Language Processing service that can be plugged and unplugged into the infrastructure.

Every service has a different format as input and output but all of them offer an interface using REST services. The advantage of doing this type of connection is that the developer is free to choose the best platform or programming language to develop each service.

Our infrastructure connects to the services using HTTP with REST calls and the message is sent in JSON format, so the modules have simple responsibilities:

- Generate a JSON with the input in the specific format required by the plugged service.
- Connect to the REST interface of the service with the required parameters such as API keys or any other authentication data.
- Receive the data from the REST service and parse it to the proper format required by the infrastructure.

The proposed framework can orchestrate new services either from commercial vendors or generated by the developer, just complying with the simple requisites described below. All services must be connected by REST calls so that the different modules are not coupled to the infrastructure. REST calls take the form of JSON messages, a de-facto standard in commercial systems.

The interaction with the container application can be done by directly calling the functions that the infrastructure offers or creating a new web service dedicated exclusively to the interaction with the infrastructure. In particular, the connection of the main modules of a conversational system works as follows.

To connect a Speech-to-Text service, the infrastructure will send to the new connector the path of the audio file. Depending on the service the connector will have to send the file in a different way. For example, Google Speech-to-Text requires a JSON message with the content of the audio file encoded in the message and IBM Watson Speech-to-Text just needs to upload the file on the request. Independently, the architecture will only require the connector to work sending the given audio file and returning the transcript text as shown in Fig. 1.

The Natural Language Understanding (NLU) service connector requires the output of the Speech-to-Text module, which is the transcribed user input. Generally, at least in the tested services (Google, Amazon and IBM), the request requires a JSON message with the user's query and the returned answer is also in JSON format. Thus, the Connector will have to send the transcribed text to the Natural Language Understanding service and this service will return a JSON with the *intents* and *entities*.

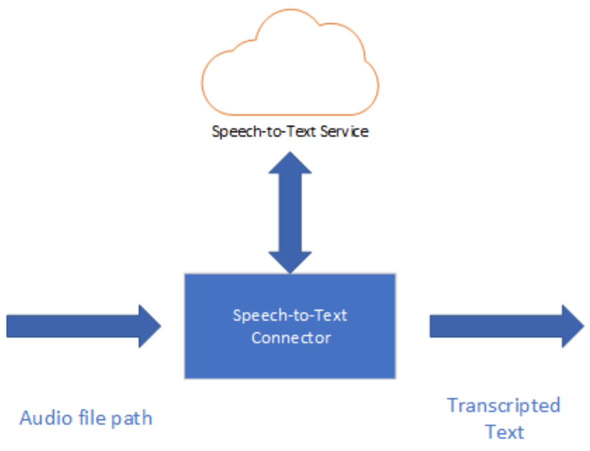

Figure 1. Scheme of the Speech-to-Text connector.

The developer will only need to implement the parser into the IH Module (see Figure 2) to generate a *registry* from the JSON answer of the NLU service.

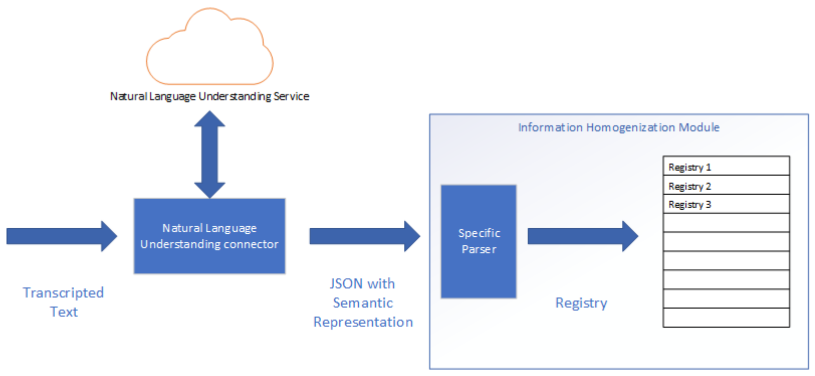
Natural Language Understanding Service
Natural Language Understanding connector
Transcripted Text
JSON with Semantic Representation
Information Homogenization Module
Specific Parser
Registry
Registry 1
Registry 2
Registry 3

Figure 2. Scheme of the Natural Language Understanding connection and the Information Homogenization Module.

After being parsed, the generated *registry* is stored in the board of the IH Module. This way, the Dialogue Manager (DM) service can retrieve from the infrastructure the *registries* that the IH Module could store in the board as well as the previous system action, to use them as a basis for decision making, as shown in Figure 3. Then the next system action and the necessary data will be returned. The next action is stored in the IH Module for the next turn of the conversation.

In some cases, the next system action may also contain the generated natural language text for the user so the usage of the Natural Language Generation (NLG) module is optional and in the case it is used it will receive the DM output and will return the natural language answer. If the developer chooses to incorporate a NLG service, the output would be the set of sentences in natural language generated as system response.

In the case of the Text-to-Speech (TTS) module, the connector receives the text to be synthesised from the infrastructure and the TTS service returns the synthesised voice

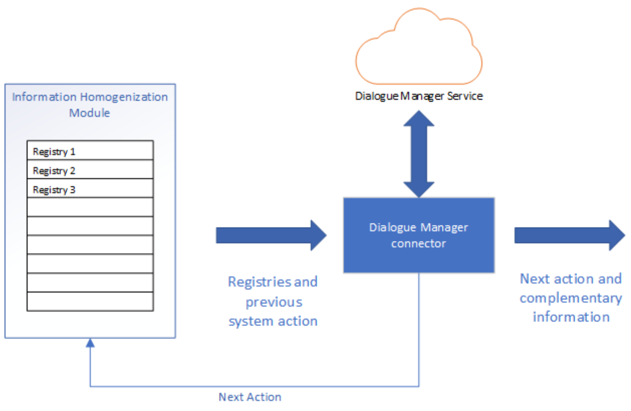
Dialogue Manager Service
Information Homogenization Module
Registry 1
Registry 2
Registry 3
Dialogue Manager connector
Registries and previous system action
Next action and complementary information
Next Action

Figure 3. Scheme of the Dialogue Manager connector.

as an audio file to be played by the container application, this can be done including the path in the JSON response (as it is done in Google Cloud) or initiating the download of the audio file as a result of the REST request (as it is the case of IBM Watson Text-to-Speech).

4. Implementation of the Architecture

The infrastructure has been implemented in Java and already incorporates connections to the online services of the most important companies. We have tested and created connectors for the following services, but the proposal is not limited to them:

- Speech-to-Text module: Google Speech-to-Text and IBM Watson Speech-to-Text.
- Natural Language Understanding: Google DialogFlow and Microsoft LUIS. We also implemented our own Natural Language Understanding service that can be plugged to the infrastructure (see Section 4.1).
- Dialog Manager: we have implemented our own dialogue manager (see Section 4.2. It is worth noticing that commercial systems often have NLU and DM linked, so it is difficult to separate understanding capabilities from the dialogue management policy.
- Natural Language Generator: we have used a template-based generation.
- Text-to-Speech: Google Text-to-Speech and IBM Watson Text-to-Speech.

To show that it is not only possible to plug-in third party services, but also to easily implement and deploy the developers' own implementations, we have implemented and offer the connector for our own NLU and DM modules, which are described below.

4.1. Sample Implementation of a Natural Language Understanding Module

Our own solution for Natural Language Understanding has been developed in Python using Okapi BM25 [13] for *intent* detection and *Term Frequency Inverse Document Frequency* (TF-IDF) for *entity* detection. BM25 is often used by search engines to detect relevant results based on the user's input and for document-scoring [14]. We have used this probabilistic function to detect the *intents* from the input, including some improvements to make the result more accurate according to observed training phrases. BM25 can be trained in a similar fashion as commercial chatbot development systems, providing a corpus with example phrases for every *intent*. When the function is applied, it computes a score for every *intent*, so the one with highest score can be selected as the most probable.

For the purpose of finding relevant keywords and *entities* in the user's input phrase we use TF-IDF. For this task, we use the same corpus employed to train BM25 grouping phrases incorporating information about *entities*. TF-IDF computes the frequency of every word in the phrase. *Entity* detection requires a higher level of post-processing for which we have applied three filters: i) list of *stopwords* that are ignored; ii) a *dictionary* of words grouped by entity types (this filter can help identifying keywords but it is not so helpful when the same keywords belong to several entity types); and iii) *aliases* and synonym detection.

4.2. Sample Implementation of a Dialogue Manager

In current toolkits dialogue management is not usually a "pluggable" service. For example, Google DialogFlow is putting the Natural Language Understanding service, the Dialogue Manager service and the Natural Language Generator together and despite the fact that it is possible to use the Natural Language Understanding part as a service, the Dialogue Manager cannot be so easily decoupled.

We developed our dialogue manager using Python following the service format described for our proposal. The main task for the manager was to direct the conversation depending on the current *intent* and the *registry* history. The decision about how to continue the conversation is taken using Sklearn library's decision trees trained using a corpus that considers not only the previous system actions, but also the confidence of the natural language understanding module.

5. Implementation of Demo Systems

As a proof-of-concept we have developed two simple pizza ordering dialogue systems for home delivery, as a sample IE service. The systems use two different configurations of the architecture, involving different services for the same tasks. The infrastructure worked perfectly and the objective to combine different services and make them work together was performed successfully.

The first Demo uses the Google Speech-to-Text service, the Microsoft LUIS Language Understanding module, the Dialogue Manager we implemented with a simple Natural Language Generator and IBM Watson Text-to-Speech. The second Demo uses IBM Watson Speech-to-Text, the Natural Language Understanding service we implemented, the same Dialogue Manager we developed with the first demo system, and Google Text-to-Speech.

A single container application that worked for both demo systems was implemented in Java. The application had a button that the user had to press to speak, then the path of the recording was given to the infrastructure and it returned the answer audio file that the Java application played. With our framework it was possible to plug different solutions for the several modules implied and the result was transparent and worked seamlessly independently of the technology used.

6. Conclusions and Future Work

This paper has introduced an architecture for the creation of modular multi-purpose conversational systems that allows developers to combine already existing commercial services with their own solutions. This way, developers can focus on the implementation of specific modules and the selection of the best-suited third-party alternatives. To generate a functional system, the developer would only need to parametrise the infrastructure part to connect with the proper REST services, and define the system *intents* and the *entities*. To show the appropriateness of the framework as a practical development solution, we have already included a repertoire of already existing solutions from Google, Amazon and IBM and also generated our own Natural Language Understanding and Dialogue Management modules successfully.

For future work we plan to retrieve the opinion of our prospective users, developers with different expertise in the development of conversational systems, to validate our proposal.

Acknowledgements

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 823907 (MENHIR project: `https://menhir-project.eu`).

References

[1] Longo F, Padovano A. Voice-enabled Assistants of the Operator 4.0 in the Social Smart Factory: Prospective role and challenges for an advanced humanmachine interaction. Manufacturing Letters. 2020;26:12–16.

[2] Adamopoulou E, Moussiades L. Chatbots: History, technology, and applications. Machine Learning with Applications. 2020 Dec;2:100006.

[3] McTear M. Conversational AI: Dialogue Systems, Conversational Agents, and Chatbots. Morgan & Claypool; 2020.

[4] McTear M, Callejas Z, Griol D. The Conversational Interface: Talking to Smart Devices. 1st ed. New York, NY: Springer; 2016.

[5] Hornos MJ, Rodrguez-Domnguez C. Increasing user confidence in intelligent environments. Journal of Reliable Intelligent Environments. 2018 Jul;4(2):71–73.

[6] Autexier S, Drechsler R. Towards Self-explaining Intelligent Environments. In: 2018 7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO); 2018. p. 1–6.

[7] Lison P, Kennington C. OpenDial: A Toolkit for Developing Spoken Dialogue Systems with Probabilistic Rules. In: Proceedings of ACL-2016 System Demonstrations. Berlin, Germany: Association for Computational Linguistics; 2016. p. 67–72.

[8] Jang Y, Lee J, Park J, Lee KH, Lison P, Kim KE. PyOpenDial: A Python-based Domain-Independent Toolkit for Developing Spoken Dialogue Systems with Probabilistic Rules. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP). Hong Kong, China: Association for Computational Linguistics; 2019. p. 187–192.

[9] Iovine A, Narducci F, Semeraro G. Conversational Recommender Systems and natural language:: A study through the ConveRSE framework. Decision Support Systems. 2020 Apr;131:113250.

[10] Nakano M, Komatani K. A framework for building closed-domain chat dialogue systems. Knowledge-Based Systems. 2020 Sep;204:106212.

[11] Bocklisch T, Faulkner J, Pawlowski N, Nichol A. Rasa: Open Source Language Understanding and Dialogue Management. arXiv e-prints. 2017 Dec;1712:arXiv:1712.05181.

[12] Kuratov Y, Yusupov I, Baymurzina D, Kuznetsov D, Cherniavskii D, Dmitrievskiy A, et al. DREAM technical report for the Alexa Prize 2019. In: 3rd Proceedings of Alexa Prize; 2019. Available from: `https://m.media-amazon.com/images/G/01/mobile-apps/dex/alexa/alexaprize/assets/challenge3/proceedings/Moscow-DREAM.pdf`.

[13] Amati G. LIU L, OZSU MT, editors. BM25. Boston, MA: Springer US; 2009. Available from: `https://doi.org/10.1007/978-0-387-39940-9\_921`.

[14] Robertson S, Zaragoza H. The Probabilistic Relevance Framework: BM25 and Beyond. Foundations and Trends in Information Retrieval. 2009 Apr;3(4):333–389.