



Pedro Elkind Velmovitsky

**iBot: An Agent-based Software Framework
For Creating Domain Conversational
Agents**

Dissertação de Mestrado

Dissertation presented to the Programa de Pós-graduação em Informática of the Departamento de Informática do Centro Técnico Científico da PUC-Rio as partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Carlos José Pereira de Lucena

Rio de Janeiro
July 2018



Pedro Elkind Velmovitsky

**iBot: An Agent-based Software Framework for Creating
Domain Conversational Agents**

Dissertation presented to the Programa de Pós-Graduação em Informática, of PUC-Rio, in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the undersigned Examination Committee.

Prof. Carlos José Pereira de Lucena

Advisor

Departamento de Informática – PUC-Rio

Prof. Ruy Luiz Milidiú

Departamento de Informática – PUC-Rio

Dr. Marx Leles Viana

Pesquisador Autônomo

Prof. Márcio da Silveira Carvalho

Vice Dean of Graduate Studies

Centro Tecnico Cientifico – PUC-Rio

Rio de Janeiro, July 5th, 2018

All rights reserved.

Pedro Elkind Velmovitsky

B.Sc. in Production Engineering, Pontifícia Universidade Católica do Rio de Janeiro 2016. He is a researcher member of the Software Engineering Laboratory at the Pontifical Catholic of Rio de Janeiro since 2016. His main studies are related to the area of software engineering, conversational agents and multiagent systems.

Bibliographic data

Elkind Velmovitsky, Pedro

iBot: An Agent-based Software Framework for Creating Domain Conversational Agents / Pedro Elkind Velmovitsky; advisor: Carlos José Pereira de Lucena. Rio de Janeiro: PUC-Rio, Departamento de Informática, 2018.

70 f. : il. (color); 30 cm

1. Dissertação (Mestrado em Informática) - Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2018.

Incluí bibliografia.

1. Informática – Teses. 2. Sistema Multiagente. 3. Engenharia de Software. I. Lucena, Carlos José Pereira de. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgements

To my parents, Márcio Velmovitsky and Fabiana Elkind Velmovitsky, my sister, Vanessa Elkind Velmovitsky, and my grandparents David Elkind Schwartz and Belinha Saubel Elkind, for the advices and encouragements throughout my life, and for always supporting me in my decisions, no matter what.

I would like to thank Professor Lucena for his expert advise and for the opportunity of learning with his experience and knowledge throughout the research process and during my Master degree.

To Dr. Marx Leles Viana, for the patience and invaluable support during the making of this work, and to Professor Ruy Luiz Milidiú, for introducing me to the subject of chatbots and encouraging me to pursue further studies in the area.

I would like to thanks all my colleagues for their wonderful collaboration.

This project would have been imposible without the support of CAPES and PUC-Rio. Thank you.

Abstract

Velmovitsky, Pedro Elkind; Lucena, Carlos José Pereira de (Advisor). **iBot: An Agent-based Software Framework for Creating Domain Conversational Agents**, 2018. 70p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Chatbots are computer programs that interact with users using natural language. Since its inception, the technology has advanced greatly and cloud-based platforms from big companies allow developers to create intelligent and efficient chatbots. However, there are not many development approaches to the main modules of a chatbot that are flexible enough to allow the creation of different chatbots for each domain, while maintaining a robust dialogue control in the application. There have been some works that try to develop a more flexible approach, each of them with their own advantages and disadvantages. One of the most notable advantages is the use of multi-agent systems to distribute and perform the tasks performed by the chatbot. In this context, this work proposes a general and flexible architecture based on multi-agent systems for building chatbots in any domain chosen by the developer, with dialogue control in the application. This architecture uses an adaptation of the information state approach, also using software agents, to perform dialogue management. To validate the proposed architecture, an user scenario involving the implementation of 4 proof of concept chatbots is analyzed discussed.

Keywords

Chatbots; Multiagent Systems; Dialogue Manager; Information State; Conversational Agents

Resumo

Velmovitsky, Pedro Elkind; Lucena, Carlos José Pereira de. **iBot: um Framework baseado em Agentes para Criar Agentes Conversacionais em Diferentes Domínios**. Rio de Janeiro, 2018. 70p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Chatbots são programas de computador que interagem com usuários utilizando linguagem natural. Desde sua origem, a tecnologia avançou significativamente e aplicações baseadas na nuvem de grandes empresas permitiram que desenvolvedores criassem chatbots inteligentes e eficientes. No entanto, não há muitas abordagens de desenvolvimento aos principais módulos de um chatbot que são flexíveis o suficiente para permitir a criação de chatbots diferentes para cada domínio, mantendo um robusto controle de diálogo na aplicação. Existem trabalhos que tentam desenvolver uma abordagem mais flexível, cada um com suas vantagens e desvantagens. Uma das vantagens mais notáveis é o uso de sistemas multiagentes para distribuir e realizar tarefas feitas por chatbots. Nesse contexto, este trabalho propõe um framework geral e flexível baseado em sistemas multiagentes para construir chatbots em um domínio escolhido pelo desenvolvedor, com controle de diálogo na aplicação. Esta solução usa uma adaptação da abordagem de estado da informação, e agentes de software, para gestão do diálogo. Para validar a arquitetura proposta, um cenário de uso com 4 chatbots de prova de conceito são analisados e discutidos.

Palavras-chave

Chatbots; Sistemas Multi-agentes; Gestão de Diálogo; Estado da Informação; Agentes Conversacionais

Content

1 Introduction	10
1.1 Problem and Proposed Solution	11
1.2 Expected Contributions	12
1.3 Research Questions	12
1.4 Organization	13
2 Background	14
2.1 Software Agents	14
2.1.1 The BDI Model	15
2.2 History and Early Chatbots	16
2.2.1 Alan Turing and the Imitation Game	16
2.2.2 ELIZA	16
2.2.3 PARRY	17
2.2.4 ALICE	19
2.2.5 Current chatbot technologies	20
2.3 Chatbot Architecture	21
2.3.1 Natural Language Understanding (NLU)	21
2.3.2 Dialogue Manager	22
2.3.3 Task Manager	27
2.3.4 Natural Language Generation	27
2.4 Speech Acts	27
3 Related Work	30
3.1 (BUCKLEY & BENZMÜLLER, 2005) and (BUCKLEY & BENZMÜLLER, 2006): An Agent-based Dialogue Management Platform - ADMP	30
3.2 (HO, NGUYEN & WOBCKE, 2006) – Smart Personal Assistant for E-Mail and Calendar Tasks	33
3.3 (LEE, LIN & WANG, 1999) – Decentralized Model and Agent Society	35
3.4 Other Works	38
4 iBot Framework: An Agent-based Software Framework For Creating Domain Chatbots	39
4.1 The Core Architecture of iBot	39
4.2 Extending iBot	41
5 User Scenario	44
5.1 Description	44
5.2 Extending iBot: Hot-Spots and Frozen-Spots	45
5.3 Login Chatbot	47
5.4 Clarification Chatbot	53
5.5 Event Chatbot	57

5.6 Comprised Chatbot	60
5.7 Discussion	63
6 Conclusion and Future Work	65
6.1 Main Contributions of the Proposed Solution	65
6.2 Main Limitations of the Proposed Approach	65
6.3 Future Work	66
References	68

List of Figures

Figure 1: Current chatbots and platforms	21
Figure 2: Linear Dialogue Script	23
Figure 3: Hierarchical State Transition Network Dialogue Model	23
Figure 4: The Architecture of ADMP	31
Figure 5: Execution of an Update Rule Agent	32
Figure 6: Architecture of SPA	33
Figure 7: Coordinator Agent Plans	34
Figure 8: Centralized model for tour information service	36
Figure 9: Agent society for spoken dialogue tour information service	36
Figure 10: iBot's Core Architecture	40
Figure 11: Extending ADMP and the Task Agent Society	46
Figure 12: NLI Module	47
Figure 13: Agents in the Login Chatbot	49
Figure 14: Behavior of <i>InputAgent</i>	49
Figure 15: Intents in the Login Chatbot	50
Figure 16: User Utterances in the <i>provideInterests</i> intent	51
Figure 17: Custom Entities in the Login Chatbot	51
Figure 18: Entity <i>interests</i> in the Login Chatbot	52
Figure 19: Login Chatbot for Company	52
Figure 20: Login Chatbot for Person	53
Figure 21: Agents in the Clarification Chatbot	54
Figure 22: Intents in the Clarification Chatbot	55
Figure 23: Custom Entities in the Clarification Chatbot	55
Figure 24: Clarification Chatbot for Beginner User	56
Figure 25: Clarification Chatbot for Experienced User	57
Figure 26: Agents in the Event Chatbot	58
Figure 27: Intents in the Event Chatbot	58
Figure 28: Custom Entities in the Event Chatbot	59
Figure 29: Event Chatbot Dialogue	59
Figure 30: Agents in the Comprised Chatbot	61
Figure 31: Intents in the Comprised Chatbot	62
Figure 32: Custom Entities in the Comprised Chatbot	62
Figure 33: Comprised Chatbot Dialogue	63

List of Tables

Table 1: Dialogue Act Recognition Example	22
Table 2: Conversational Act Descriptions in SPA	33

1 Introduction

A chatbot can be defined as a human-computer dialogue system (CAHN, 2017) or, in other words, a computer program that interacts with users using natural language (WOUNDERBERG, 2014) (GATTI de BAYSER & CAVALIN, 2017).

The concept of a chatbot can be first attributed to Alan Turing, who wrote his seminal work about machine intelligence and thinking in 1950 (TURING, 1950). In fact, it was a chatbot program, called ELIZA, that first passed a version of the Turing Test in 1966 by simulating the behavior of a Rogerian psychologist (WOUNDERBERG, 2014). With advances in natural language understanding (NLU) and machine learning (ML) techniques chatbot technology has evolved since those days (APPEL, 2018), as evidenced by a surge in the development of increasingly intelligent software in research and in business.

The adoption of the technology by the public has also increased, in part through the use and popularity of smartphone based chatbots, such as Siri and Google Now — as well as products such as Amazon's Alexa — and in part due to the launch of chatbot hosting platforms by Facebook, Slack, Skype, Telegram, among others (CAHN, 2017).

However, even though there have been great advancements in NLU, as evidences by cloud-based solutions such as DialogFlow (“Dialogflow – Basics”, 2018) and IBM Watson (WHITE, 2018), applications are still far from perfect, as most of them generate good results in specific domains; a general model for interpreting each and every utterance, independent of context, is still unattainable. Therefore, to deliver the best experience for the user, developers must consider building their chatbots using an efficient architectural solution that ensures the best solution for their domains.

With this, considering the complexity and interaction of the different components involved in building a chatbot, a promising approach is the use of multi-agent systems (MAS) in their development, especially in performing different tasks.

While there has been a lot of research integrating dialogue systems and chatbot architecture with multi-agent systems, most of these architectures usually deal with one specific domain, or few correlated domains, with some works trying to increase the domains handled by distributing tasks to specific agents. However, an emerging challenge comes considering these works are not flexible enough to allow for the development of chatbots in different domains, or do not provide a robust dialogue control throughout the application.

In response to this challenge, we present an agent-based Software Framework for Creating Domain Conversational Agents (iBot). It is a general and flexible framework based on multi-agent systems, which will facilitate building chatbots in any domain chosen by the developer, with dialogue control in the application. This approach aims at providing new resources for the developer creating domain conversational agents. As such, more human characteristics can be considered in order to improve the deliberation process. By using this framework, it is possible to build chatbots that: (i) distribute specific tasks to software agents, increasing the system's intelligence and (ii) use the information state approach to dialogue management, thus allowing modularity, reuse and dialogue control.

1.1 Problem and Proposed Solution

There are several works which present the main architecture and modules of chatbots (WOUNDERBERG, 2014), (CAHN, 2017), (BUCKLEY & BENZMÜLLER, 2005). However, there are not many development approaches to these modules that are flexible enough to allow the creation of different chatbots for each domain — it is important to note that here the word “domain” is referring to the specific tasks that the agent is trying to accomplish; for instance, a tourist information service agent, may perform tasks related to obtaining information about buses or about the weather —, while maintaining a robust dialogue control.

There have been some works that try to develop a more flexible approach, most notably (HO, NGUYEN & WOBCKE, 2006), which proposes a centralized architecture in which a coordinator mediates communication between the user and task agents, and (LEE, LIN & WANG, 1999) proposes a decentralized model that the dialogue flow is controlled by each task agent. These models consider the use of agent technologies to implement the tasks that must be performed by the chatbot,

providing greater flexibility. However, it is difficult to develop robust different domain chatbots using these approaches, mainly due to difficulties in adding new tasks and maintaining dialogue control throughout the application, respectively (the related work will be expanded upon in Chapter 3).

With this, it was noted a need to develop a general approach for the development of chatbots, focusing on flexibility. Therefore, the proposed solution presented in this work has the goal of providing a general framework, called iBot, for the development of different chatbots. The Core Architecture of the framework does not make any assumptions about which platform or programming language the developer must use, but rather provides a blueprint for developers to create chatbots in different domains, while maintaining a level of dialogue control in the application. This Architecture can be implemented in a number of ways, generating different instantiations of the iBot framework that best suit the developer's needs.

To validate the iBot framework, a user scenario involving the implementation of 4 proof of concept chatbots using its Core Architecture is discussed. For this validation, the programming language Swift was used, and the chatbots were developed for iOS platforms.

1.2 Expected Contributions

This section describes the main expected contributions of the work:

- Literature Overview of the concepts and state-of-the-art involved in building chatbots, in order to examine the problem and allow the development of the solution;
- Development of the Core Architecture of the iBot framework;
- Extension of iBot for validation, through the analysis of an user scenario involving the implementation of 4 proof of concept chatbots in iOS using the framework.
-

1.3 Research Questions

The main research question is:

- (RQ) How chatbots for different domains can be developed, while maintaining dialogue control in the application?

Sub-questions are:

- (RQs1) How does a chatbot work?
- (RQs2) What are the main elements that compose a chatbot?
- (RQs3) What are the available approaches to manage a dialogue?
- (RQs4) What are the advantages and disadvantages of each approach?
- (RQs5) How does one use the information state approach with the goal of maximum reuse?
- (RQs6) Does the proposed solution allow for the development of chatbots for different tasks?
- (RQs7) Does the proposed solution allow dialogue control in the application?

The first five sub-questions will be answered by reviewing literature, specially presented in Chapter 2, while the last two sub-questions will be answered through the development and validation of chatbots using the proposed framework.

1.4 Organization

The dissertation is organized as follows:

- Chapter 2 describes the concepts of software agents and chatbots;
- Chapter 3 details the related work;
- Chapter 4 describes the proposed solution;
- Chapter 5 expands upon the proof of concept for validating the solution and describes its implementation;
- Chapter 6 presents some conclusions and future work.

2 Background

This Chapter describes the main characteristics of software agents and the BDI model. In addition, also providing background on chatbots' early history and implementation, chatbot architecture and the concepts of speech acts as defined in the Theory of Language.

It will start by providing an overview of multi-agent systems and the modelling of the behavior of software agents using the BDI model. Then, it will describe the beginnings of chatbots and dialogue systems, followed by a description of the architecture of chatbots and dialogue systems, as well as providing a detailed view of ADMP, an Agent-based Dialogue Management Platform. Finally, it will introduce the concepts of speech acts.

2.1 Software Agents

Software agents are reactive systems that exhibit some degree of autonomy in achieving a goal: an agent is capable of independent action in unpredictable and changing environments, without the need of direct human intervention. The system is called an "agent" because it is action oriented: an agent should actively pursue its goals and tasks, independently reasoning about the best way to do so. Therefore, agents can adapt to current circumstances, dynamically changing their behavior to fully achieve its desires and intentions.

An agent is not usually found alone in a system; in fact, individual agents interact with each other, collaborating to perform complex tasks and achieve their respective objectives. This constitutes a multi-agent system (MAS).

Multi-agent systems increase modularity and diminish the complexity of software engineering problems, by providing a modelling of systems as a set of entities/organizations who interact between themselves and with other entities to achieve their goals and objects.

Among the main characteristics of a software agent, we can cite:

- **Autonomy:** agents must act without direct human intervention, according to its reasoning;
- **Reactivity:** agents perceive their environment and respond to changes in it;
- **Pro-activeness:** agents should be opportunist and goal-oriented, looking to execute actions when applicable to their goals;
- **Social:** agents are capable of interacting with other agents, when appropriate, in order to achieve their goals.

2.1.1 The BDI Model

There are many ways to model the reasoning and behavior of agents. The most popular and researched approach is the BDI (Belief-Desire-Intention) model. To talk about this model, according to (BORDINI, HÜBNER & WOOLDRIDGE, 2007), we need to address the idea that we can talk about computer programs as if they had a “mental state”. Thus, when we talk about a belief–desire–intention system, we are talking about computer programs with computational analogues of beliefs, desires and intentions. These are described below:

Beliefs: are information the agent has about the environment. This information, however, is something the agent believes in but it may not be necessarily true. As an example, an agent may perceive from its environment the fact that it is raining. However, the rain may stop before the next reasoning cycle of the agent – in this case, his belief is outdated and incorrect.

Desires: are the possible states of affairs that the agent might like to accomplish. That does not mean, however, that the agent will act upon it – it is a potential influencer of the agent’s actions.

Intentions: are the state of affairs that the agent has decided to act upon. In other words, intentions can be considered as a selected option between the potential set of options/desires that the agent has decided to pursue.

These are the key data structures of the BDI model. The decision-making approach used by the agent, therefore, is practical reasoning: the agent weighs conflicting information for and against the available options, according to its beliefs and desires. The result of this deliberation is the adoption of intentions, which in turn will lead to the execution of actions.

2.2 History and Early Chatbots

Chatbots are computer program that interacts with users using natural language (WOUNDERBERG, 2014) (GATTI de BAYSER & CAVALIN, 2017). This sub-chapter discusses the origins of the concept, history and techniques involved in the first chatbots, and the state-of-the-art for current chatbot technologies.

2.2.1 Alan Turing and the Imitation Game

In 1950, Alan Turing published his seminal work about machine intelligence and thinking (TURING, 1950). To answer the question of whether machines can think Turing proposed a test which he called the "imitation game" — nowadays known as the Turing Test — in which an interrogator asked questions to a human and to a machine, aiming to identify which of the two is the machine. If the interrogator is unable to do so, it is established that the machine can think. By proposing a machine that can dialogue with humans in natural language, Turing was creating the very concept of a chatbot.

2.2.2 ELIZA

In 1966, researchers at the MIT created ELIZA, the first chatbot to pass a version of the Turing Test. ELIZA (WEIZENBAUM, 1976) (WOUNDERBERG, 2014) simulated a Rogerian psychologist whose goal was to make people reflect about their current situation by using techniques to keep the patient talking. Patients could not tell that they were talking to a program, to the surprise of ELIZA's creator, Joseph Weizenbaum.

ELIZA was able to carry on conversations such as:

Patient: Men are all alike.

ELIZA: IN WHAT WAY

Patient: They're always bugging us about something or other.

ELIZA: CAN YOU THINK OF A SPECIFIC EXAMPLE

Patient: Well, my boyfriend made me come here.

ELIZA: YOUR BOYFRIEND MADE YOU COME HERE

Patient: He says I'm depressed much of the time.

ELIZA: I AM SORRY TO HEAR YOU ARE DEPRESSED

Patient: It's true. I am unhappy.

ELIZA: DO YOU THINK COMING HERE WILL HELP YOU NOT TO BE UNHAPPY

Patient: I need some help, that much seems certain.

ELIZA: WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP

Patient: Perhaps I could learn to get along with my mother.

ELIZA: TELL ME MORE ABOUT YOUR FAMILY

To perform this dialogue, ELIZA used simple pattern matching and substitution techniques to process the input and produce an appropriate output. Some examples are as follows (WOUNDERBERG, 2014):

** you are (depressed|sad) * => I AM SORRY TO HEAR YOU ARE \I*

** all * => IN WHAT WAY*

** always * => CAN YOU THINK OF A SPECIFIC EXAMPLE*

Outside of the realm of Rogerian therapy, ELIZA also performed well. In one case, a business executive tried to contact a programmer via teletype but, unbeknownst to him, got ELIZA instead. The dialogue went on for a while, until the executive phoned the programmer asking him why he was being "snotty" (WOUNDERBERG, 2014). This can also be considered an example of ELIZA passing a somewhat modified version of the Turing Test, since the executive did not realize he was talking to a machine — even though he was not trying to distinguish between a computer and a human, like in the Turing Test.

2.2.3 PARRY

In 1972, the psychiatrist Kenneth Colby created PARRY (SHIEBER, 1994) (WOUNDERBERG, 2014). Although this chatbot works similarly to ELIZA, using pattern matching techniques to produce a suitable output, PARRY had the goal of simulating a paranoid schizophrenic. Implementing a crude model of behavior and a conversational strategy, it was more advanced than its predecessor. Colby described it as "ELIZA with an attitude". PARRY also passed a variation of the Turing Test, in which psychiatrists were given transcripts of dialogues with PARRY and transcripts with actual paranoid schizophrenic patients, and were asked

to decide which one was simulated. The psychiatrists did no better than random guessing in this test (SHIEBER, 1994).

The following is an example of a dialogue with PARRY (WOUNDERBERG, 2014).

Psychiatrist: Good afternoon. My name is Doctor Jones.

PARRY: Pleased to meet you.

Psychiatrist: How are you?

PARRY: All right I guess.

Psychiatrist: Could you tell me your name?

PARRY: Pat Smith

Psychiatrist: How old are you?

PARRY: 28.

Psychiatrist: I thought you might have some problems.

PARRY: I've been upset lately.

Psychiatrist: By what?

PARRY: People get on my nerves sometimes.

Psychiatrist: Tell me more.

PARRY: Do you know anything about bookies?

Psychiatrist: No. What about bookies?

PARRY: A bookie didn't pay me once.

Psychiatrist: When did this happen?

PARRY: A couple of months ago.

PARRY and Eliza actually met several times, most notably at the ICC 1972 conference in Washington, DC. An interesting demonstration followed, and a part of the dialogue between them is listed below ("RFC 439 - PARRY encounters the DOCTOR", 2018):

ELIZA: Good Evening. Tell me your problems. Please terminate input with a period or a question mark.

PARRY: People get on my nerves sometimes.

ELIZA: I am not sure I understand you fully.

PARRY: You should pay more attention.

ELIZA: Suppose you should pay more attention.

PARRY: You're entitled to your own opinion.

ELIZA: What makes you think I am entitled to my own opinion?

PARRY: It bothers me just to be around people in general.

ELIZA: Please go on.

PARRY: I went to the track at Bay Meadows a while back.

ELIZA: What does that suggest to you?

PARRY: I went to the races.

ELIZA: What else would you like to discuss?

PARRY: Horseracing interests me.

2.2.4 ALICE

ALICE is a chatbot created in 1995 by Dr. Richard Wallace. It has won the Loebner Contest — an annual competition to identify the most "human" computer and to award \$100.000,00 for the first program that passes an unrestricted Turing Test — in 2000, 2001 and 2004.

ALICE was inspired by ELIZA, and uses an XML Schema called AIML (Artificial Intelligence Markup Language) for specifying patterns and responses (WOUNDERBERG, 2014). For example:

```
<category>
<pattern>WHAT IS YOUR *</pattern> <template>Guess!</template>
</category>
```

This pattern matches inputs such as "What is your name?" or "What is your favorite movie".

Among several interesting features in AIML, such as the use of variables and the capability of maintaining context, is recursion. For example:

```
<category>
<pattern>DO YOU KNOW WHAT * IS</pattern>
<template><srai>WHAT IS <star/></srai></template>
</category>
```

In this case, the phrase "Do you know what Computer Science is?" will redirect to the category "What is Computer Science", and to the appropriate response.

2.2.5 Current chatbot technologies

Nowadays, chatbot technology has become very popular due to its integration in smartphones and smart devices (WEINBERGER, 2017). Examples of modern chatbots are Siri, embedded in Apple's iOS-based devices; Google Now, in Android devices; Amazon's Echo and Alexa; and Microsoft Cortana (GATTI de BAYSER & CAVALIN, 2017). These bots take advantage of advanced natural language understanding and machine learning techniques to generate responses based on analysis of web search results. Other modern chatbots use Statistical Machine Translation techniques to "translate" input into output responses.

Big players in the technology market, such as Google, Microsoft and IBM, have launched cloud-based platforms — such as Google Cloud, Microsoft Azure and IBM Cloud, respectively — allowing access for developers to their services and solutions. Several of these deal with machine learning training problems, including natural language processing – such as DialogFlow, Google's platform for processing natural language utterances and developing intelligent conversational agents.

Some applications, such as Facebook, Slack, Skype, Telegram, among others, allow chatbots to be hosted and deployed. Facebook Messenger, for example, had 34.000 developers on its platform and was hosting 30.000 bots in the end of 2016 (CAHN, 2017).

This allows chatbots developers to have great efficacy and efficiency in developing and deploying intelligent chatbot applications for specific domains and releasing them in different platforms. Figure 1 shows some examples of current chatbots and platforms available.



Figure 1: Current chatbots and platforms

2.3 Chatbot Architecture

This sub-chapter answers the research sub-questions **#RQs1**: How does a chatbot work? and **#RQs2**: What are the main elements that compose a chatbot?

Building a chatbot requires several components, some of them specific to the domain and tasks being handled. These chatbots systems usually have common modules. However, there are different approaches to how they should be developed and implemented, the concepts remain the same.

This section introduces these concepts involved in creating a chatbot dialogue system.

2.3.1 Natural Language Understanding (NLU)

The first step when a user produces an utterance is receiving and processing it. If the utterance is spoken, a Speech Recognition module will identify the spoken words, producing an unstructured representation of them (CAHN, 2017). If the dialogue system's input format is text only, a Speech-To-Text conversion is not needed.

The unstructured text — whether a result of direct user input in case of text-only interfaces or output from a Speech Recognition module — is then fed into the Natural Language Understanding module. This component receives the text and

outputs a semantic representation of its content, that can be read and understood by the Dialogue Manager component.

There are several approaches in implementing NLU. One common way is using a frame-slot semantics, based on a semantic grammar, where input sentences by the user contain information that fill the appropriate slot. Another way is recognizing Dialogue Acts, which represent the function of a text/sentence. Table 1 depicts examples of Dialogue Act Recognition (CAHN, 2017). Machine learning techniques can be used to perform this classification, such as decision trees and neural networks.

Dialogue Acts are usually specific to a certain domain. So, instead of asking the general function of an utterance (whether the sentence is a question or a statement, for example), it can be interesting to discover what is the user's intention behind said utterance — for instance, the user may want to book a flight or schedule an appointment.

Dialogue Acts which are domain-specific are called intents (CAHN, 2017). Intent Identification is a very popular approach in cloud-based platforms for Natural Language Understanding, such as DialogFlow or Microsoft Azure.

Table 1: Dialogue Act Recognition Example

Speaker	Dialogue Act	English
A	Conventional-opening	Hallo!?
B	Conventional-opening	Hi Peter!
B	Statement	It's me, Michael.
B	Question	How are you?
A	Conventional-opening	Hello Michael!
A	Statement	Very well.
A	Question	And you?
B	Statement	I'm well too.

2.3.2 Dialogue Manager

This section answers the research sub-questions **#RQs3**: “What are the available approaches to manage a dialogue?”; **#RQs4**: “What are the advantages and disadvantages of each approach?”, and **#RQs5**: “How does one use the information state approach with the goal of maximum reuse?”.

The Dialogue Manager is the central component of a chatbot. It is responsible for controlling the state and flow of the conversation, storing discourse context and managing the different components in the architecture. There are several distinct approaches to implementing a Dialogue Manager (WOUNDERBERG, 2014), such as finite-state based systems, frame-based systems, agent-based systems and plan-based systems. A different approach to dialogue management, called the information-state approach, generalizes the previous approaches by viewing dialogue management in terms of information state updates.

Finite-state based systems

In finite-state based dialogue managers the dialogue is specified as a set of states, with transitions denoting several paths through a graph (WOUNDERBERG, 2014). State Transition Networks, which there is branching of paths, Hierarchical State Networks, which dialogue fragments can be reused, such as subroutines in a software, and Augmented Transition Networks, which includes storing and retrieving of information based on the user input — can be used to increase the efficacy of this approach (BICKMORE & GIORGINO, 2006). Figure 2 shows an example of a linear finite-state dialogue, while Figure 3 shows a Hierarchical State Network, with branching and use of subroutines — the square boxes in the Figure 3 (BICKMORE & GIORGINO, 2006).

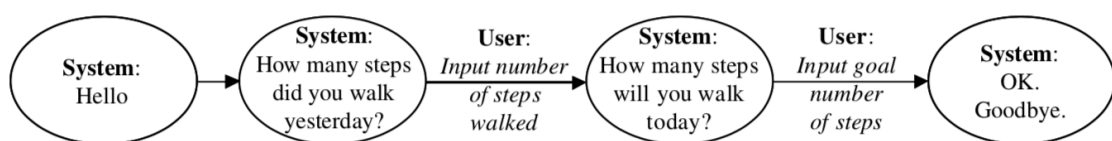


Figure 2: Linear Dialogue Script

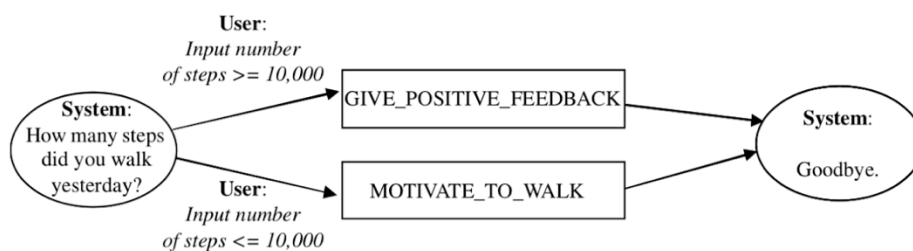


Figure 3: Hierarchical State Transition Network Dialogue Model

Frame-based systems

Frame-based systems work as a frame-slot procedure. The system asks questions to the user, filling the necessary slots of information to perform a task. The dialogue flow is not fixed, as in finite-state based systems, but depends on the content and order of the user input. In fact, frame-based systems can behave such as finite-based systems. See the following example:

System: What is your destination?

User: London

System: What day do you want to travel?

User: Friday

In this case, where each slot is filled one at a time, the behavior of the system is similar to a finite-based approach. Now, let's suppose the following dialogue:

System: What is your destination?

User: London on Friday around 10 in the morning.

System: I have the following connection...

In this dialogue, the system receives all the information at once and fill the slots. If no more information is necessary, the system can perform the task of finding a flight according to the user's parameters.

Plan-based systems

This approach is based on the principle that humans who communicate have underlying intentions, related to achieving a goal; user input is seen, in this approach, as speech acts (WOUNDERBERG, 2014). For more information on speech acts, see Session 2.5.

In the plan-based approach, the system should be able to reason and infer about the user's goal during the dialogue. Inferring a goal is necessary because what the user says may not be directly related to his intent — this phenomenon is known in Theory of Language as indirect speech acts (BICKMORE & GIORGINO, 2006).

For example, a speaker may say to the listener "It's very hot in here", so that the listener opens the window.

So, plan-based systems assume that each act is part of the plan, and its objective is to infer the user's goals and perform specific tasks to achieve it. (HO, NGUYEN & WOBCKE, 2006) implements an architecture involving plan-based systems, executed by a software agent.

Agent-based systems

In these systems, a conversation is seen as an interaction between two agents, each one being capable of reasoning about its own actions and beliefs, and about the actions and beliefs of the other agents participating in the dialogue.

It is important to note that the word "agent", in this case, means modelling the participants as autonomous entities capable of reasoning about its own behavior and that of others, not the technology involved in multi-agent systems.

This approach is more suitable for mixed initiative dialogue, allowing both the user and the system to take control of the dialogue and introduce new topics.

Information State based systems

The term information state is used to define information about the conversation that is stored by the system. More specifically, (LARSSON & TRAUM, 2003) defines the information state of a dialogue as the information necessary to distinguish it from other dialogues, representing the cumulative additions from previous utterances and motivating the selection of future action.

This approach proposes a unifying view of dialogue management, in which independent dialogue theories can be implemented and evaluated in a reusable foundation. Similar names for information state are "conversational score", "discourse context" or "mental state".

The Information State approach provides a method for specifying a theory of dialogue (BUCKLEY & BENZMÜLLER, 2005), which consists of:

- An information state;
- Representations for the information state;
- Dialogue moves;

- Update rules;
- Update strategy.

The information state, as mentioned, is the description of the state of the discourse and its participants. It stores dialogue level knowledge, such as questions and answers, beliefs of the user, beliefs of the system, the last utterance and who performed it, among others.

For each aspect of the dialogue context that is modelled, a representation must be chosen. This representation can range from simple structures such as a list or a string, to more complex representations such as an attribute-value matrix.

Dialogue moves provide an extension from the notion of speech acts to include aspects of the relationship of the speech act to the rest of the dialogue. When a dialogue move is performed, its content may result in a change being made to the state of the dialogue.

As the dialogue progresses, the information state must be updated to reflect the consequences/effects that actions of the participants have on the dialogue context. In order to perform these updates, update rules are implemented to be fired in reaction to observed dialogue moves. These rules are specified by precondition rules and effects: preconditions define which information state is active at a time, and effect rules indicate the changes that must occur to achieve the new information state. Update rules may also have side conditions, allowing external functions to be called within the rule to calculate the transition.

To control how updates are made, an update strategy must be declared. This strategy is an algorithm which decides the update rules that should be fired at a given time, such as selecting the first applicable rule.

(BUCKLEY & BENZMÜLLER, 2005) and (BUCKLEY & BENZMÜLLER, 2006) implement a dialogue manager that uses the information state approach together with the software agent paradigm: each precondition rule is associated with an agent, that evaluates if the rule is applicable at a certain time and performs the necessary transition calculations. An update agent, controlling the update strategy, is responsible for checking the available transitions and selecting one to become the next information state. This architecture, called the Agent-based Dialogue Management Platform (ADMP), is expanded in Chapter 2.4.

2.3.3 Task Manager

The term Task Manager is used generically here to indicate the component responsible for actually performing the necessary tasks requested by the user, such as booking a flight or retrieving information about the weather.

2.3.4 Natural Language Generation

The Natural Language Generation component gets a meaning representation of what to say from the Dialogue Manager and conveys this in natural language to the user, whether in speech or text. ELIZA's AIML schema is an example of this module, using a template-based approach. Nowadays there are more advanced techniques for Natural Language Generation, including using Statistical Machine Translation techniques to generate responses (CAHN, 2017).

If the dialogue system is implemented with spoken dialogue as output, a Text-To-Speech component is necessary to take the generated text and transform it into synthetic speech. A common method of implementing Text-To-Speech is concatenating prerecorded speech together, forming utterances (WOUNDERBERG, 2014).

2.4 Speech Acts

A speech act is a term created by language philosophers denoting the sense in which utterances do not just provide meaning, but they also perform action (LEVINSON, 2016). By talking, a person is performing intentional goal-directed speech acts.

In other words, speech act theory does not subscribe to the Aristotelian view that sentences and propositions are either true or false. John Austin was one of the first philosophers to break this notion by working out that "an utterance in a dialogue is a kind of action being performed by the speaker" (WOUNDERBERG, 2014). For example, when the proper authority utters "I name this ship the Titanic", the state of the world changes: the name of the ship is, from now on, Titanic. Therefore, this speech act, in a sense, is no different from any other action. Austin called these kind of speech acts *performatives*: acts which incite action, changing

one's reality. In contrast, a *constantive* is defined by Austin as an utterance which merely describes a situation.

In terms of the composition of speech acts, Austin claimed that any utterance can be divided into three parts:

Locutionary act: the act of saying something; the actual utterance of the sentence. It includes a phonetic act, involving the production of noises; a phatic act, involving the production of words specific to a certain vocabulary and grammar; and a rhetic act, involving the product of the phatic act with a context or reference (TRAUM, 1999).

Illocutionary act: acts performing in saying something, for example asking or answering a question and giving information.

Perlocutionary act: acts performed by saying something. It is related to intentions by the speaker that influence the hearer's thoughts, feelings and actions. Persuasion, surprise and deterrence are examples of a perlocutionary act.

Indirect speech acts, as mentioned in sub-chapter 2.3.2, are also defined by Austin.

John Searle, another philosopher of language, extends Austin's work, particularly in relation to illocutionary acts. He classifies these acts into 5 categories (WOUNDERBERG, 2014):

Assertives: commit the speaker to something being true; related to the beliefs of the speaker. Examples are suggesting, boasting, concluding, among others.

Directives: attempts by the speaker to convince the listener of doing something; related to the desires of the speaker. Examples are asking, ordering, requesting, advising, among others.

Comissives: commit the speaker to future action; related to the intentions of the speaker. Examples are promising, vowing, betting, among others.

Expressives: express the internal state of the speaker. Examples are thanking, apologizing, welcoming, among others.

Declaratives: bring about a different state of the world, such as the Titanic example mentioned previously.

There are many reasons to consider these theories on language and utterances while developing dialogue systems. They can be very useful, for example, in AI models. However, more interesting correlations appear when we consider multi-agent systems.

Software agents are, by definition, action-oriented instead of knowledge-oriented: agents must decide on the best course of action available at the time, even if their knowledge is incomplete. Since speech acts define talking as performing an action, conceptually there is a great correlation between these theories and conversational agents.

Practically, these correlations become even greater. KQML (Knowledge Query and Manipulation Language), which proposes to be a universal communication language for agents, is based on Austin's performatives. (TRAUM, 1999) offers a consolidation of the use of speech act in AI models, as well as an interesting discussion on the validity and formalization of this theory in multi-agent communication.

Furthermore, the first 3 types of illocutionary acts described by Searle, namely *assertives*, *directives* and *comissives*, are also related to beliefs, desires and intentions, the key data structures of the BDI model. In fact, (GATTI de BAYSER & CAVALIN, 2017) states that “if speaking consists in making one's intentions to make a certain speech act explicit, then speaking may be seen as a communication of intentions” and that in BDI “an intentional action is a function to accomplish a desired goal and is based on the belief that the course of action will satisfy a desire”. So, in the context of dialogue an intent is a goal expressed in an utterance, and by recognizing the intent a chatbot can choose the correct dialogue flow for responding to it – in fact, this is the approach used for the proof of concept chatbots, described in Chapter 5, which use DialogFlow to obtain information about the utterance’s intent.

Although a philosophical discussion on the validity of speech act theory in dialogue systems is not on the scope of this work, one cannot deny the attractiveness of speech acts in multi-agent systems, since it allows "an agent theorist or designer to place agent communication within the same general framework as agent action" (TRAUM, 1999). Therefore, even though this work will deal with a chatbot architecture based on multi-agent systems and not with the structure of dialogues, it is interesting to keep speech act theory concepts in mind when developing dialogue systems.

3

Related Work

This Chapter describes the works that helped during the conceptualization of the problem and the creation of the proposed solution. In sub-chapter 3.1, an agent-based dialogue management platform (ADMP) using the Information State approach will be described. This platform is used in iBot to perform dialogue management. Sub-chapters 3.2 and 3.3 describe works that helped to conceptualize and contextualize iBot, analyzing the different approaches used in these works to create distributed architectures with MAS, it was possible to develop a generic framework for the development of different domain chatbots. Lastly, sub-chapter 3.4 presents works that provided an invaluable literature overview of the main concepts of conversational agents.

3.1 (BUCKLEY & BENZMÜLLER, 2005) and (BUCKLEY & BENZMÜLLER, 2006): An Agent-based Dialogue Management Platform - ADMP

One popular approach to design autonomous agent systems is using a blackboard mechanism, which is a global data store for sharing information. This approach is used, for example, in the Open Agent Architecture (COHEN, et al., 1994). In it, agents can communicate with each other directly or read and write events on the blackboard (such as their goals, or task specialties). One disadvantage of this method, however, is that because the model is highly distributed and the control flow depends largely on the interactions between agents, it is difficult to guarantee robust behavior and dialogue control.

(BUCKLEY & BENZMÜLLER, 2005) and (BUCKLEY & BENZMÜLLER, 2006) implement and instantiate the Dialogue Manager using ADMP (Agent-based Dialogue Management Platform), an architecture which uses a society of software agents to compute and control update rules and information state transitions by using a blackboard. Because it provides a meta-level to reason about heuristics,

ADMP provides better control of the dialogue. Figure 4 shows the ADMP architecture.

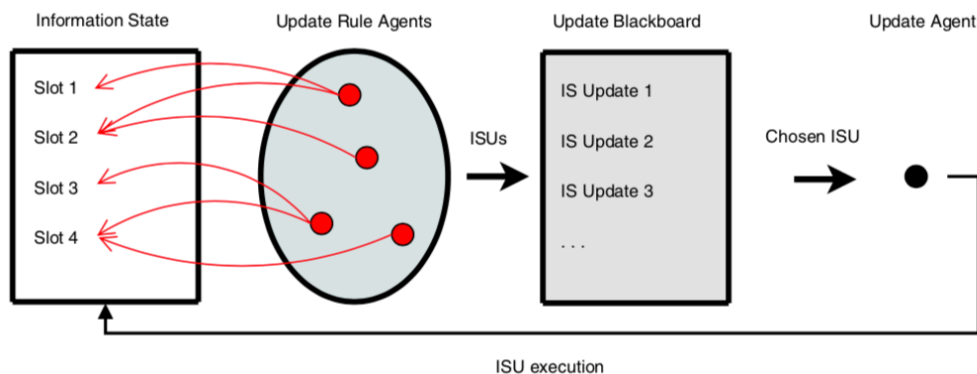


Figure 4: The Architecture of ADMP

On the left is the *Information State*, the central data structure of the system. It is made up of slots, which store a value. These slots are read by software agents, called *Update Rule Agents* (URA). Each of these agents, in turn, are associated with an update rule, which have in its preconditions a subset of the set of slots in the information state. When the update rule agent observes that its preconditions hold, it computes the information state update encoded in the rule and writes the result to the update blackboard.

In Figure 4, The *Update Agent* (UA) that surveys the update blackboard. After a timeout, or some other stimulus, it chooses the heuristically preferred *IS Update*, executes it, and resets the system for a new turn.

In order to make the repeated checking of the values in the information state slots by the agents more efficient, time stamping is applied to represent when a slot was last updated. Each slot has a timestamp, which is set to 0 when the slot is created. Whenever a slot is updated, its timestamp is incremented by 1, allowing the agents to determine whether the value contained in the slot has changed since the last time it was observed: if an agent observes that the timestamp changed, it knows that an update has been made on the slot, which leads the agent to test the preconditions of the rule. If the preconditions are satisfied, *side conditions* are executed (such as calling an external system), and finally the effects are evaluated and the new information state computed. Figure 5 depicts the execution loop of an *Update Rule Agent*.

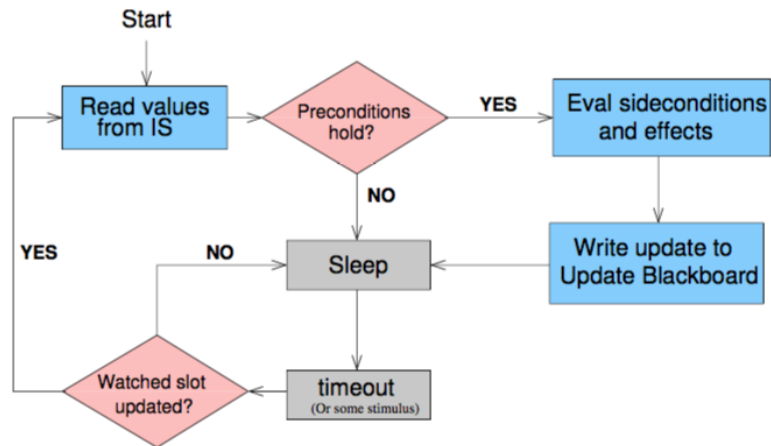


Figure 5: Execution of an Update Rule Agent

It is important to note that the agents run concurrently, so that many agents may be simultaneously computing results or may have made calls to external systems — it is up to the update agent to determine which computed new information state to use.

The proposed solution presented in Chapter 4 uses ADMP to perform dialogue management based on the information state approach. ADMP was chosen for the solution because it is based on the information state approach, thus allowing the implementation and evaluation of different theories of dialogue. Also, it does not constraint types and values in the slots of the information state, leaving the decision of what to model in the dialogue, i.e., which information will be important in the context of the domain, to the developer. This is particularly important considering that the proposed architecture needs to be flexible enough to allow for the development of applications in different domains. Each of these domains, in turn, may need different information to be modelled in the information state, and ADMP provides this flexibility. Finally, ADMP uses multi-agent technology to implement the information state approach. Since the proposed architecture uses software agents to increase modularity, in order to better perform complex tasks, the use of a multi-agent system in dialogue management provides an interesting complement and a nice fit.

3.2 (HO, NGUYEN & WOBCKE, 2006) – Smart Personal Assistant for E-Mail and Calendar Tasks

(HO, NGUYEN & WOBCKE, 2006) discusses the architectural aspects of a *Smart Personal Assistant* (SPA) system to help users with e-mail and calendar tasks. In this model, a special Coordinator agent mediates communication between the user and specialist task agents, responsible for handling e-mail and calendar tasks. Figure 6 depicts the overall architecture of this SPA.

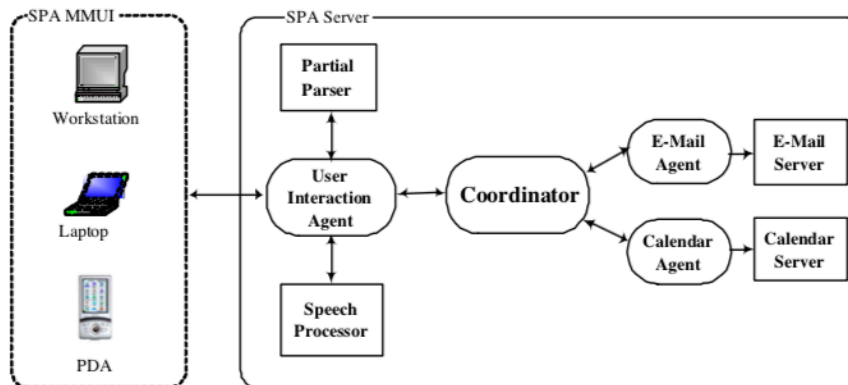


Figure 6: Architecture of SPA

The *Coordinator*, is responsible for both dialogue modelling and coordination. It is implemented as a BDI agent that uses plans to coordinate the action of the individual assistants and for encoding the system's dialogue model. This dialogue model is based on the theory of speech acts by Searle, where it is assumed that each utterance has a main speech act, known as a conversational act in this context. The conversational acts interpreted by the system are shown in Table 2.

Table 2: Conversational Act Descriptions in SPA

Conversational Act	Act Description
Request	ask the addressee to perform a domain task
Respond	describe the result to the hearer
Clarify	ask the addressee to clarify ambiguities
Greet	express the speaker's greetings/feelings
Confirm	clarify ambiguities by expressing agreement/disagreement
Ack	express an acknowledgement

To interpret each conversational act in an utterance and coordinate with the task specific agents, the coordinator is specified with a set of plans divided into four groups: conversational act determination and domain task classification, intention identification, task processing and response generation. Figure 7 presents the *Coordinator Agent* and when each plan is activated – shown in the Figure in the dotted boxes.

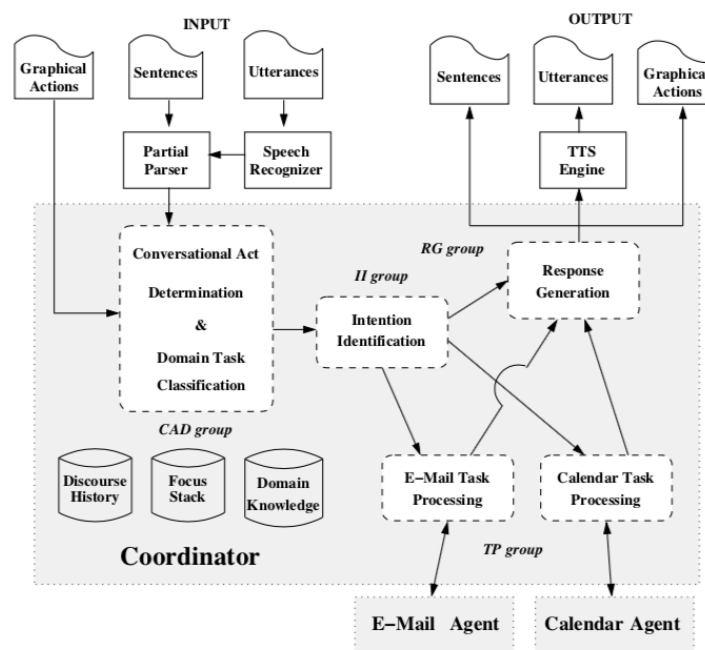


Figure 7: Coordinator Agent Plans

While this SPA architecture is interesting for two reasons. First, by delegating the tasks performed by the system to specific agents, who have complete autonomy in achieving their goals, it diminishes the complexity of the system and can increase the domains handled by it. Also, by centralizing dialogue and coordination in the Coordinator agent, it maintains a robust dialogue control of the application. It is important to note, however, that the SPA architecture also has its shortcomings; mainly, every change in the system will require the creation of more plans in the agent's knowledge base.

First, not all plans can be reused; only domain independent plans related to the dialogue model can be reused in different domains. Regarding the domain-related plans, the Coordinator needs to know everything about the task agents, including their capabilities, beforehand, in order to call these agents through plans in the Coordinator's database. Therefore, adding new abilities to these agents includes generating new plans for the Coordinator.

Also, adding new agents that may perform other tasks may be extremely difficult, requiring a remodeling of the plans for coordinating each agent in the Coordinator; therefore, augmenting the system's capability of handling more domains may not be a trivial task.

Lastly, if we want to increase or change the system's dialogue model, to include more conversational acts for instance, more plans will also be created. So, for each change in the system, the Coordinator becomes bigger and more complex. Analyzing the advantages and disadvantages mentioned above, the proposed solution presented in the next chapter hopes to maintain the centralized aspect of the SPA architecture by implementing a centralized and complex Dialogue Manager component based on ADMP and the information-state approach – as opposed to the plan-based Coordinator. By doing so, the proposed solution maintains a centralized structure for obtaining dialogue control while implementing a Dialogue Manager that can be easily altered or modified with the addition of new agents or dialogue models. Regarding dialogue models, if developers want to add new speech acts to the system, or use a different dialogue theory altogether, in terms of the information state this means only the remodeling of its slots to retrieve the relevant information in the dialogue – while a plan-based system would require the creation of new set of plans. Moreover, the task agents in the system, as will be explained in Chapter 4, will be called as side conditions of the update rules in ADMP, like any other module in the system – the difference being that the agents have the intelligence and autonomy to pursue their own goals using the MAS paradigm. So, to include a new task agent in the system, developers will only need to add new update rules and agents to the system to execute this rule, instead of writing new plans in the Coordinator.

3.3 (LEE, LIN & WANG, 1999) – Decentralized Model and Agent Society

Another interesting decentralized architecture is presented in (LEE, LIN and WANG, 1999). This paper first describes a centralized model for multi-agent and multi-domain spoken dialogue systems, as shown in Figure 8, which illustrates a tour information dialogue system.

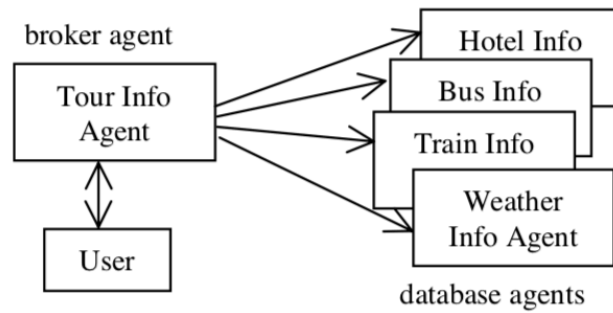


Figure 8: Centralized model for tour information service

In this model, the broker agent understands the user's request, sends queries to each agent according to the user's intent, gets the answer and generates a response to the user. While this model may be effective for single-turn dialogue, it is not very effective when considering multi-turn: the broker agent must be extremely complicated, switching the dialogue smoothly between different domains, and storing the dialogue state and knowledge. Therefore, adding new domains will be complicated in this architecture, increasing the complexity of the broker agent.

To solve these inefficiencies, the work proposes a different architecture in which the dialogue state and knowledge, as well as the control flow of the dialogue, are stored in each of the task agents, called here Spoken Dialogue Agents (SDA), when they are “activated”. Figure 9 shows this system.

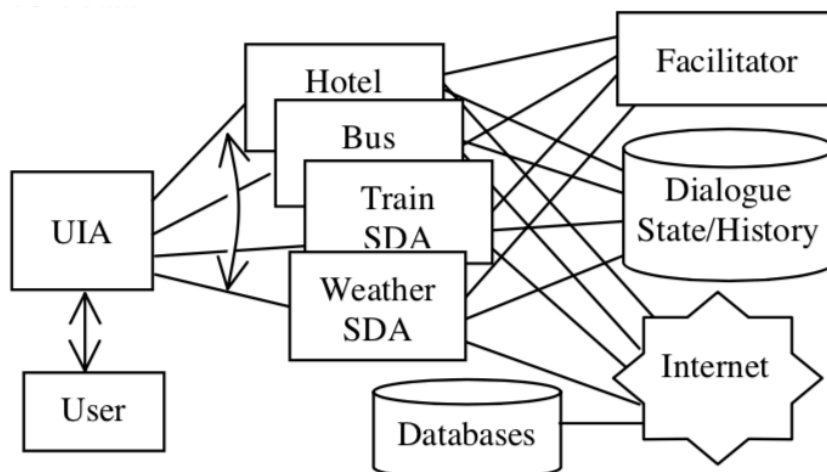


Figure 9: Agent society for spoken dialogue tour information service

In this architecture a domain switching protocol is implemented, where the Facilitator, deciding that the user's utterance is from a different domain (using Graph Search for the different domains), switches from the current SDA to the SDA that can handle the utterance. When this switch occurs, the State Dependent Data

(SDD), mainly the dialogue state and history, is transferred to the new agent, who has complete responsibility regarding its tasks and its dialogue.

The architecture proposed in (LEE, LIN & WANG, 1999) has many advantages. First, it is relatively simple to add new and correlated domains to the system, since each agent is responsible for the conversation flow in their turn, performing their tasks and having access to the SDD. Also, the user is not aware of the switch between agents, since the UIA (User Interface Agent) provides one uniform interface for communication.

However, the main disadvantage of this architecture is that it gives up completely any kind of centralized dialogue control. Suppose we are implementing a dialogue system in a critical domain, such as healthcare; some control of the information obtained by the system through its interaction with the user must be centralized in a component, in order to guarantee that all information required for a task is collected, with quality, by the system.

While (LEE, LIN & WANG, 1999) focuses on multi-domain applications, and this work's goal is to develop a framework for chatbots in different general domains, it still provides some valuable lessons and insights. First, the idea of distributing tasks to specialist agents, also present in the SPA architecture from the previous section, is very attractive.

Moreover, one of the goals of the proposed solution is to maintain dialogue control in the application, something that does not happen in the decentralized model mentioned above. Therefore, the iBot framework presented in the next chapter presents a more centralized architecture, which minimizes the disadvantages of the centralized tour information service of Figure 8 by developing a more complex dialogue management component, based on ADMP and the information state approach, instead of using a single agent to perform all coordination, interpretation and interaction functions. In fact, the proposed solution's design is very similar to the model in Figure 9, except for the inclusion of the Dialogue Manager and for the specialist task agents, which do not have the autonomy to actually perform dialogues in iBot. Another difference is that dialogue level knowledge is stored in the Dialogue Manager, instead of being transferred by SDA. We hope that by combining the best of the two models we can achieve an interesting solution for developing chatbots in different domains, while maintaining dialogue control.

3.4 Other Works

(WOUNDERBERG, 2014), (CAHN, 2017) and (BUCKLEY & BENZMÜLLER, 2005) provide a literature overview both of the basic concepts and the state-of-the-art, of building dialogue systems. They also present a general architecture for these systems, as mentioned in Chapter 2.3.

(WOUNDERBERG, 2014), in particular, implements a Dialogue Manager using the information state approach in TrindiKit, a Prolog toolkit, coupled with pattern matching and substitution techniques. This Dialogue Manager is used in an application consisting of a statistics tutor.

These works, by presenting and analyzing the history, main concepts, methods and components involved in creating state of the art dialogue systems, were of great help. In particular, their description of the interaction between the main components involved in building chatbots, as described in Chapter 2, served as a great introduction to the subject and increased the author's understanding of the necessary efforts and concepts to actually construct dialogue systems. Because of this, they provided invaluable references to this work.

4 iBot Framework: An Agent-based Software Framework For Creating Domain Chatbots

This Chapter described the proposed solution to the problem described in sub-chapter 1.1: iBot, an agent-based software framework for creating domain chatbots while maintaining dialogue control in the application.

Combining the advantages of the related works mentioned in Chapter 3 while diminishing the disadvantages, the proposed solution uses ADMP and the information-state approach to create a complex and centralized Dialogue Manager component that can be easily expanded and modified to store dialogue level knowledge. By distributing tasks to software agents to increase modularity in the system, the proposed solution facilitates the creation of chatbots in different domains that have a robust dialogue control in the application.

In sub-chapter 4.1, the Core Architecture of iBot is described. As mentioned in Chapter 1, this Architecture does not make any assumptions about which platform or programming language the developer must use, providing a blueprint for developers to create chatbots. This Architecture can be implemented in a number of ways, generating different instantiations of the iBot framework that best suit the developer's needs. Sub-chapter 4.2 describes how to extend the Core Architecture for instantiation of the framework.

4.1 The Core Architecture of iBot

Examining the previous concepts and architectures it is clear that distributing complex tasks in an agent society, which can autonomously realize specific tasks, can increase modularity and the overall power of the system by allowing more domains to be handled. Also, some control in the dialogue is desirable, guaranteeing quality in the information and robustness of behavior.

Therefore, we propose a software framework for building chatbots. This framework allows the creation of intelligent chatbots that are able to perform complex tasks and maintain dialogue control while being flexible enough to allow

the development of software in different domains. Figure 10 depicts the Core Architecture of the iBot framework.

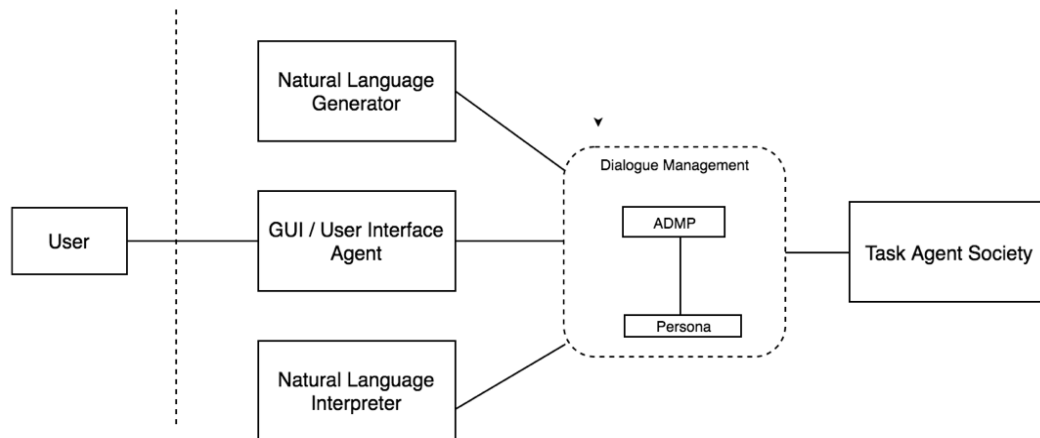


Figure 10: iBot 's Core Architecture

The user interacts with the application through a GUI, or the *User Interface Agent*. The application may deal with different correlated domains, such as the tour information service mentioned in the previous Chapter, then, the GUI will allow the user to seamlessly switch between different applications (HO, NGUYEN & WOBCKE, 2006). The architecture also accounts for *Natural Language Interpreter* and *Generator* components, to process the user's input according to the goals of the system and generate output.

On the center of the architecture is the *Dialogue Management* component, which is based on the information state approach implemented in ADMP, as explained in sub-chapter 3.1. ADMP, by distributing the management of the dialogue using software agents to complement the information state approach, make for a modular and relatively simple Dialogue Manager — for instance, in contrast to the plan-based Coordinator in (LEE, LIN & WANG, 1999) presented in sub-chapter 3.3.

By using the optional *Persona* module, developers may want to establish a persona in their program. For instance, (WALLIS, 2005) implements Eugene the Cuttlefish, a conversational agent that is vain and likes to be paid compliments about his colors; he may blush or get angry, and even withdraw for the conversation. In order to implement personalities like Eugene's, the *Persona* component can make alterations to the beliefs of the update agent in ADMP, altering its behavior.

The *Task Agent Society* represents a multi-agent system with software agents capable of performing the tasks in the domain. To implement a tour information service, for example, agents responsible for obtaining information about hotels, buses or the weather would be part of this society. They would also be able to access internal or external in order to accomplish their goals.

In a dialogue using this architecture, the user communicates with the *GUI*. The user input is passed to the *Dialogue Management* component, updating the information state. According to update rules, this component calls the *Natural Language Interpretation* module, which processes the input and extracts the relevant information from it, updating the relevant slots in the information state. These updates trigger the firing of new update rules, which may lead to other modules or components being called in turn. For example, if a slot in the information state stores the user's intention, and this is recognized as being "Obtain Bus Information", the update rule related to this slot will fire, and the necessary tasks accomplished. Those tasks may be to call the agent in the *Task Agent Society* responsible for obtaining information about buses. Once this information is returned, the update rule agent will place it on the blackboard, and the update agent will select it as the next dialogue move to be performed. The *Natural Language Generator*, then, generates a natural language output to provide the information that the user requested.

4.2 Extending iBot

It is important to note that the Core Architecture is not executable. To create an application using iBot, developers need to extend its main components to instantiate the framework. Therefore, they will need to do the following:

- (i) **Natural Language Interpreter:** to extend the Natural Language Interpreter component, developers have several alternatives. They may use their own personalized techniques, such as adapting the classic pattern matching/substitution technique described in Chapter 2, or use a number of cloud-based solutions, such as Microsoft Azure, Google's DialogFlow and IBM Watson, which allows the training of Machine Learning models with samples of utterances expected in the dialogue. Also, it is assumed that this component encapsulates a

speech-to-text converter, if the input accepts spoken utterances. The next chapter will present how the proof of concept chatbots extend this component using DialogFlow.

- (ii) **GUI:** the GUI, or User Interface Agent, can be extended in a number of ways, representing the interface for communicating with the user. If the chatbot is in a multi-modal application, then the GUI may be a number of screens, such as computer or mobile screens. The next chapter will detail how the proof of concept chatbots use the screen from a mobile device, such as an iPhone, to communicate with the user in the application.
- (iii) **Natural Language Generator:** to extend this module, developers may use a number of techniques. The responses may be hardcoded in the code – which is the approach used by the proof of concept chatbots – or be generated by more advanced methods, such as Statistical Machine Translation (CAHN, 2017).
- (iv) **Dialogue Management:** this is the most critical component of iBot, together with the Task Agent Society. To extend this model, the only compulsory condition is to instantiate the elements in ADMP, meaning the Update Rule Agents, the Update Agents and the Information State and its Updates. The developer may choose the IDE to create MAS that best suit him, and create and store the Information State according to the system’s requirements. While ADMP purposely does not require that the Information State is in one format, it does provide some basic rules, which can be seen at (BUCKLEY & BENZMÜLLER, 2005). The next chapter describes how the proof of concept chatbots implement the agents and Information State in ADMP using Swift, the IDE XCode and the *iMobile* framework for developing multi-agent systems in iOS (MIRANDA & LUCENA, 2017). While the proof of concept chatbots does not require them to have a persona, developers may also extend this module, for example by creating new beliefs for the agents in ADMP or in the Task Agent Society. The authors in (ALVES, VIANA & LUCENA, 2018) present personality traits by adapting the BDI model for normative multi-agent systems.

- (v) **Task Agent Society:** to extend the Task Agent Society module, developers must create software agents to perform the tasks necessary for the chatbot in the application. These agents are to be called by the Update Rule Agents as side conditions in the Update Rules. The next chapter describes the creation of the Task Agent Society using *iMobile*.

5 User Scenario

This Chapter presents the user scenario for the validation of the iBot framework with the implementation of 4 proof of concept chatbots for iOS platforms.

It starts by describing the scenario and the chatbots, detailing the implementation details and results of each chatbot in the following sub-sections, and concludes with a discussion about the use and validation of iBot in the creation of the proof of concepts.

5.1 Description

This work has been developed jointly with LES PUC-RJ - Laboratório de Engenharia de Software PUC-RJ (Laboratory of Software Engineering PUC-RJ), which in turn has partnerships with IDOR - Instituto D'Or de Pesquisa e Ensino (Institute D'Or of Research and Teaching), a non-profit organization whose goal is to promote technological advancement in the healthcare area.

One of these partnerships proposes the creation of a mobile app to stimulate scientific dissemination. Most specifically, the app has three main stakeholders: listeners, researchers and institutions. Researchers and institutions use the app to find each other and organize scientific events — for example, a university may wish to create an event about astronomy, while astronomers may want to share their knowledge with the public; so, they use the app to connect and create the event. Listeners, on the other hand, attend the events that they are interested in.

The idea of the app came from the success of worldwide events such as Pint of Science, and research showed that the Brazilian people is interested in science — 61% of the interviewed declared being interested by the subject, a percentage larger than the one from the European Union — while they lack knowledge on the subject — 87% did not know the name of a Brazilian scientific institution, while 94% does not know the name of a Brazilian scientist (“Unicamp”, 2017).

Considering the basis and the idea for the app, and during talks with LES and IDOR, four different chatbots were conceptualized, to help the user of the app:

- First: Chatbot to help in the filling of login forms;
- Second: Chatbot to help users understand about the subject being exposed in an event;
- Third: Chatbot to help users find events according to their interests and to confirm their presence;
- Fourth: Lastly, a chatbot comprising all functions above.

The chatbots were developed for iOS using Apple's IDE XCode and the programming language Swift 3.3.

5.2 Extending iBot: Hot-Spots and Frozen-Spots

Hot-Spots: to implement each of the proof of concept chatbots, it is necessary to instantiate the iBot framework by extending its hot-spots in the core components of the architecture. The hot-spots for this implementation are: (i) **ADMP**, including the Update Rule Agents; (ii) the agents in the **Task Agent Society**, and (iii) **Natural Language Interpreter**, implemented as a DialogFlow application for each chatbot, detailing the user's expected utterances and their underlying intention.

Regarding (i) and (ii), Figure 11 shows the extension of the ADMP and the Task Agent Society for each of the proof of concept chatbots. These agents will be expanded in the following Chapter for each chatbot. The Comprised Chatbot, comprises the Update Rule Agents and the Task Agent Societies of previous chatbots, plus an additional Update Rule Agent, as will be described sub-chapter 5.6.

The agents were implemented using the *iMobile* framework for developing multi-agent systems in iOS (MIRANDA & LUCENA, 2017). Therefore, extension points present in this framework, mainly the ones for extending the *Agent* and *Behavior* classes, were used. It is important to note that there are no restrictions in terms of the MAS creation frameworks and their respective extension points in iBot.

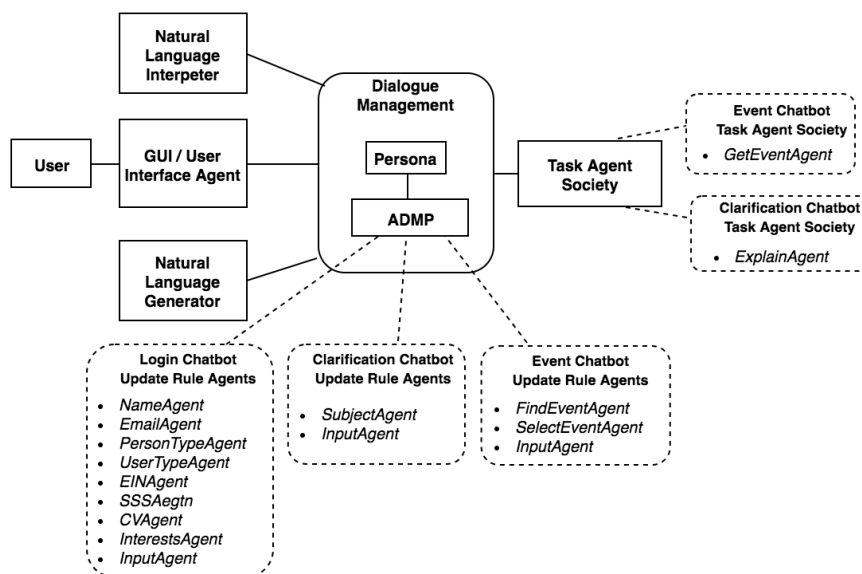


Figure 11: Extending ADMP and the Task Agent Society

As for (iii), to interpret the user’s natural language utterances, Google Cloud’s Dialogflow platform was used. DialogFlow allows the training of Machine Learning models according to samples of user’s utterances. It also provides an iOS SDK, allowing the developer to make a request to DialogFlow. This request returns the user’s intentions and parameters embedded in the utterance. Figure 12 shows the *NLI* module. It makes a request to a DialogFlow application, created for each of the chatbots, in the function *performQuery* – declared in line 24. It also analyses if the intent of the user is “quit_bot”. This intent is identified when the user says utterances like “Goodbye” or “See you later”. In this case, the chatbot is closed in the app.

Lines 38 and 39 update the Information State slots *intent_list* and *DFResponse* directly, without using the blackboard. This is for two reasons. First, it is assumed in the implementation that NLI will always be called in the beginning of the turn, and the following update rules will be based on the slots updated at this moment. Second, these slots are related to maintaining context by storing the dialogue’s history; therefore it makes sense that these values will always be updated. So, to simplify the NLI component directly updates the necessary slots.

The respective DialogFlow applications will be detailed further in the following Chapters for each chatbot.

```

15 let nli = NLI()
16
17 class NLI {
18   class var sharedInstance: NLI {
19     return nli
20   }
21
22   var delegate: NLIDelegate?
23   // Function to handle request from DialogFlow
24   func performQuery(senderId:String,name:String,text:String) {
25
26     let request = ApiAI.shared().textRequest()
27     if text != "" {
28       request?.query = text
29     }
30     else {
31       return
32     }
33     request?.setMappedCompletionBlockSuccess({ (request, response) in
34       let response = response as! AIResponse
35       InformationState.sharedInstance.dfResponse.append(response)
36       print("Action identified: ", response.result.action)
37       // Update Intent List
38       InformationState.sharedInstance.intent_list.0.append(response.result.action)
39       InformationState.sharedInstance.intent_list.1 += 1
40       // Check if action is quit_bot, and exits the chatbot
41       if response.result.action == "quit_bot" {
42         if let parameters = response.result.parameters as? [String:AIResponseParameter]
43         {
44           {
45             if (parameters["quit_bot"]?.stringValue) != nil
46             {
47               let deadlineTime = DispatchTime.now() + .seconds(2)
48               DispatchQueue.main.asyncAfter(deadline: deadlineTime, execute: {
49                 self.delegate?.quit()
50                 print("Entered here")
51               })
52             }
53           }
54         }, failure: { (request, error) in
55           print(error)
56         })
57       ApiAI.shared().enqueue(request)
58     }
59 }

```

Figure 12: NLI Module

Frozen-Spots: the frozen-spots for this implementation are: (i) **GUI**, in this case a screen in an Apple mobile device, such as an iPhone (ii) **Natural Language Generation**, in this case the chatbot accepts written and speech input, using native iOS development classes to generate natural language speech.

All chatbots also have an Update Agent, which continually checks the blackboard for Information State Updates and chooses the preferred one, according to some heuristic.

5.3 Login Chatbot

The first proof of concept chatbot that was implemented is the login chatbot. This program allows users to log in the app, storing their name and email. If the user is a company, then it should ask for the Employer Identification Number. If the user is a person, then it should ask for more information, such as Social Security number, Interests and link to the person's CV (if the person is a researcher).

The Information State for the login chatbot is defined as follows:

Non-Domain Related

- Utterance_List;

- *DFResponse*;
- *Intent_List*;
- *Next_Utterance*.

Domain Related

- *Name*;
- *Email*;
- *Id*;
- *Interests*;
- *cvLink*;
- *User_Type*;
- *Person_Type*.

The *Utterance_List* slot stores the list of utterances spoken in the dialogue, and who spoke it. The *DFResponse* slot stores the current response from *DialogFlow*, *Intent_List* stores the list of intents expressed in the dialogue and *Next_Utterance* store's the system's next utterance, sent to the GUI. The rest of the slots are domain related: *name* stores the username, *email* stores the user's email, *id* stores the user's Social Security or Employer Identification Number (depending on whether the user is a person or a company), *Interests* store the user's interests, *cvLink* stores the link to the user's CV, *user_type* stores if the user is a person or a company and, lastly, *person_type* stores if the user is a researcher, listener, or both.

The Update Rule Agents are defined according to what intent they are monitoring. For example, *NameAgent* monitors if the current intent is "provideName", and *EmailAgent* monitors if the current intent is "provideEmail". The only URA related to a non-domain related slot is the *InputAgent*, which calls the *Natural Language Interpreter* when a new utterance is spoken by the user.

Figure 13 shows these agents. Since the login chatbot does not need to perform any tasks outside collecting and storing user data, it does not use any other agent than the ones present in the ADMP architecture.

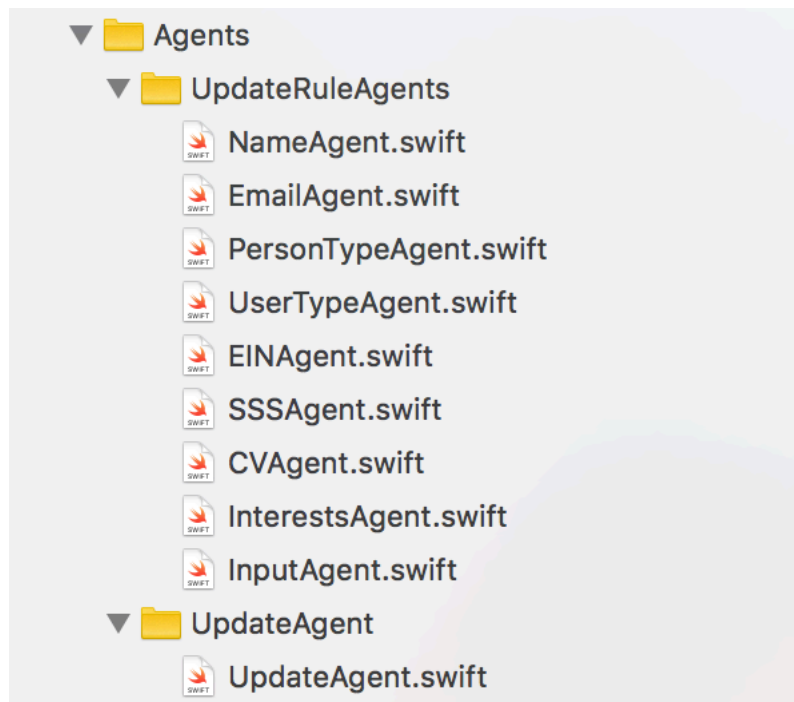


Figure 13: Agents in the Login Chatbot

Figure 14 shows the code for the behavior of *InputAgent*. Line 10 imports

```

9  import Foundation
10 import ApiAI
11
12 class InputAgentBehaviour: CyclicBehaviour {
13     var isUpdates: InformationStateUpdate? = nil
14     var intentTimestamp = 0
15     var currentMsg = 0
16
17     override init(agt: Agent)
18     {
19         super.init(agt: agt)
20     }
21
22     override func action() {
23         print("Input Agent monitoring subject")
24
25         while(isUpdates == nil) {
26             isUpdates = preconditionRule()
27         }
28
29         print("Publishing input IS Update to Blackboard")
30         UpdateBlackboard.sharedInstance.isUpdates.append(isUpdates!)
31         isUpdates = nil
32     }
33
34     func preconditionRule() -> InformationStateUpdate? {
35         if (intentTimestamp != InformationState.sharedInstance.utterance_list.1 ) {
36             intentTimestamp += 1
37             NLI.sharedInstance.performQuery(senderId: "userId", name: "userName" , text:
                 InformationState.sharedInstance.utterance_list.0)
38         }
39         return nil
40     }
41 }

```

Figure 14: Behavior of *InputAgent*

the DialogFlow API, called *ApiAI* (which was the original name of the DialogFlow tool). In Line 22, the function *action()* defines the behavior of an agent. It continues on a loop until its precondition, defined in the function *preconditionRule()* in line

34, returns an Information State Update – defined as a tuple vector of type $(String: Any)$, where the *String* field is the name of the slot in the information state and *Any* is the new value of the slot in the update. In this case, the only precondition is that the timestamp for the *Utterance_List* slot of the information state has changed — meaning that the user has made a new utterance — checked in line 35. If the timestamp changed, the *InputAgent* calls the *Natural Language Interpreter (NLI)* module in line 37.

The other URA are similar to this agent, with the main difference being that they check the timestamp of the *Intent_List* slot and the value of the last intent to see if it matches the required intent.

Figure 15 shows Dialogflow’s interface for this chatbot, with the respective intents. Each intent has samples of user’s utterances, providing examples of what the user might say in a conversation.

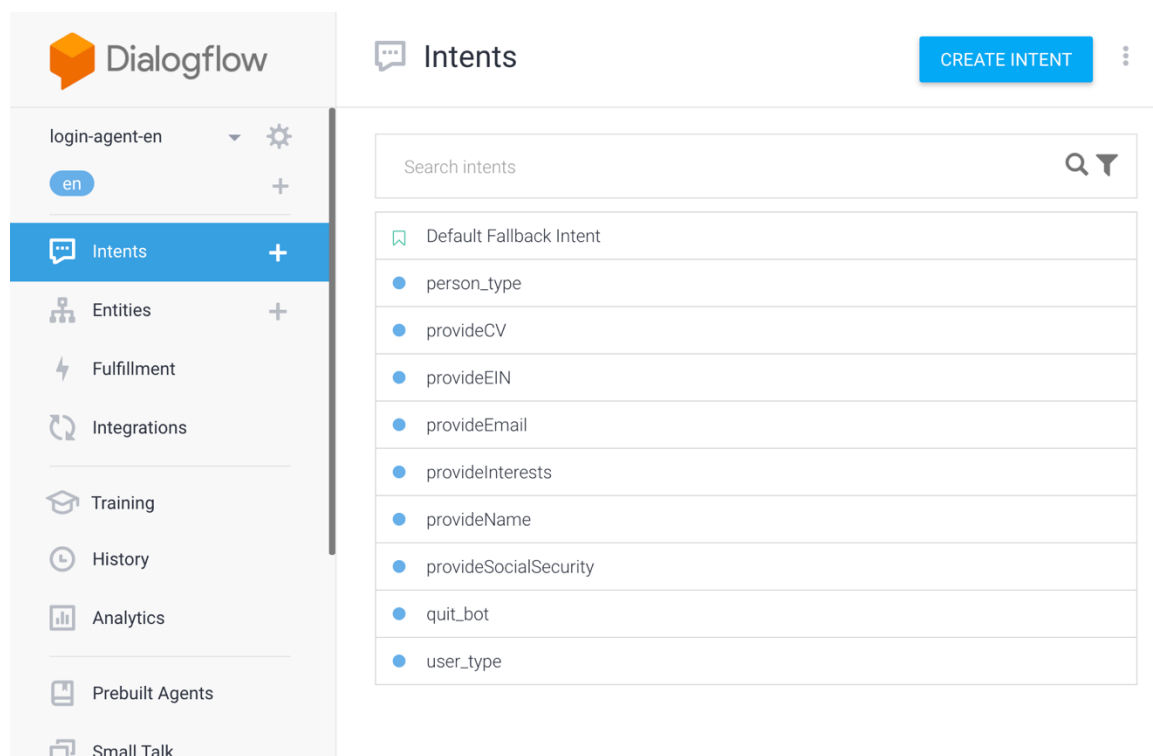


Figure 15: Intents in the Login Chatbot

Figure 16 shows examples of these utterances for the intent *provideInterests*.

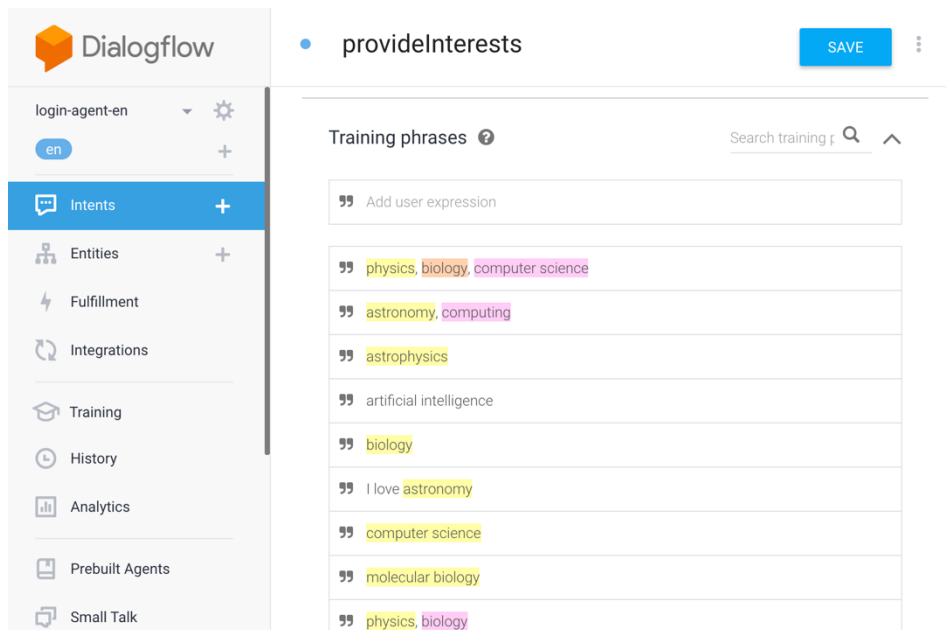


Figure 16: User Utterances in the *provideInterests* intent

The colored elements in the phrases represent entities. *Dialogflow* provides some custom utterances to identify common elements on a dialogue, such as date and time, and also allows developers to create their own customized entities. Figure 17 shows the entities for the login chatbot, and Figure 18 details the entity *interest*, to identify interests expressed in the dialogue.

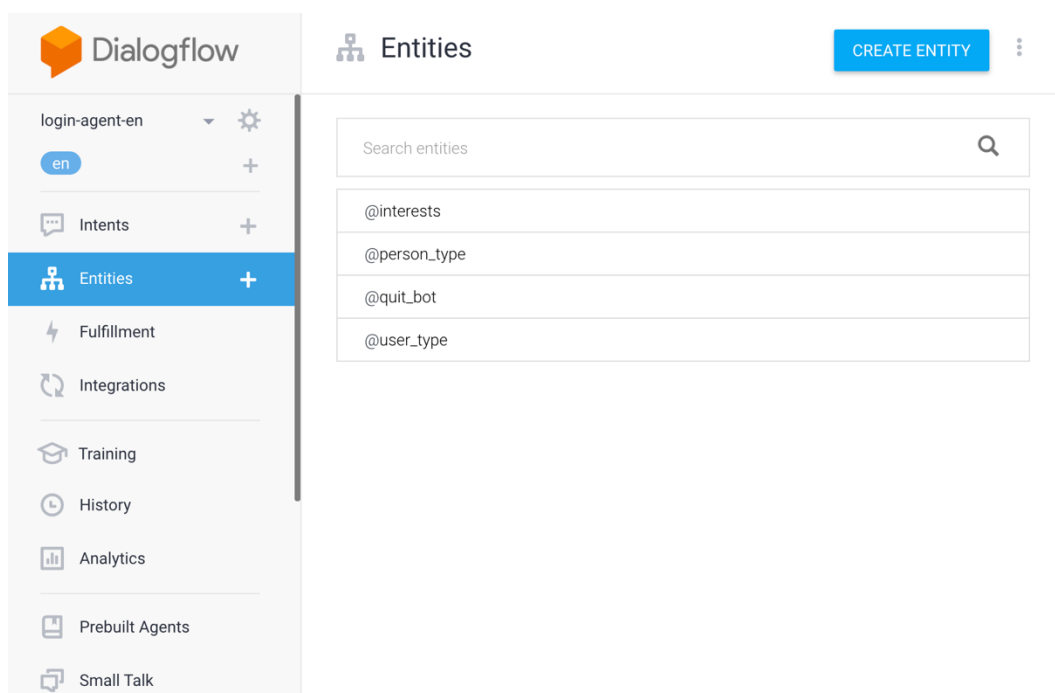


Figure 17: Custom Entities in the Login Chatbot

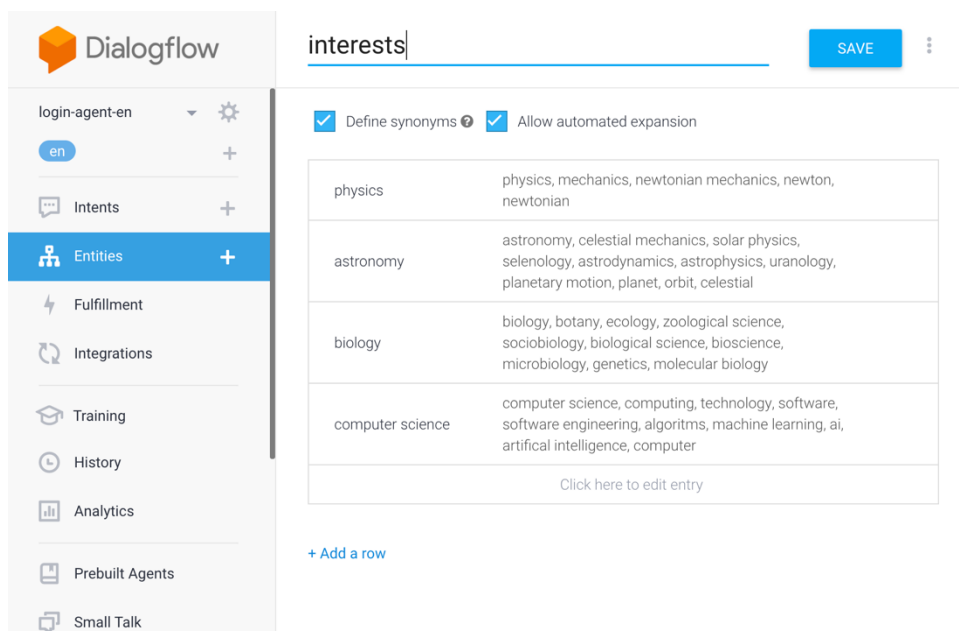


Figure 18: Entity *interests* in the Login Chatbot

Figure 19 shows the Login Chatbot for a dialogue in which a company is performing the login.

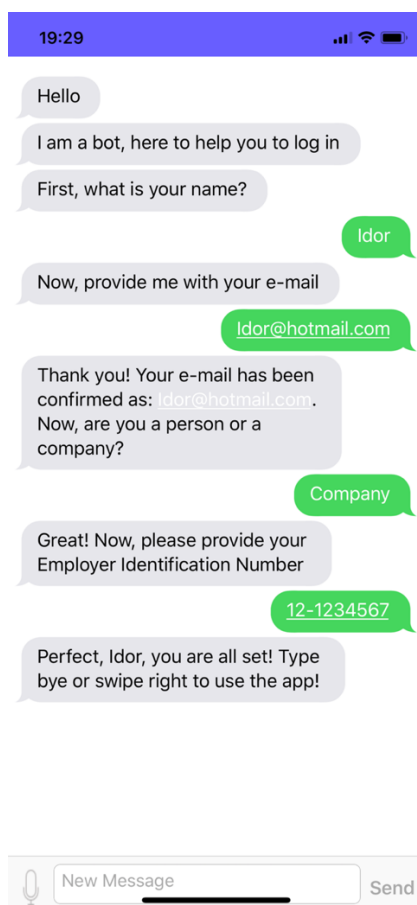


Figure 19: Login Chatbot for Company

Figure 20 shows the dialogue in which a person is logging in. The complete dialogue of the login chatbot can be seen in the following link: <https://youtu.be/BFpEnit6VOU>.

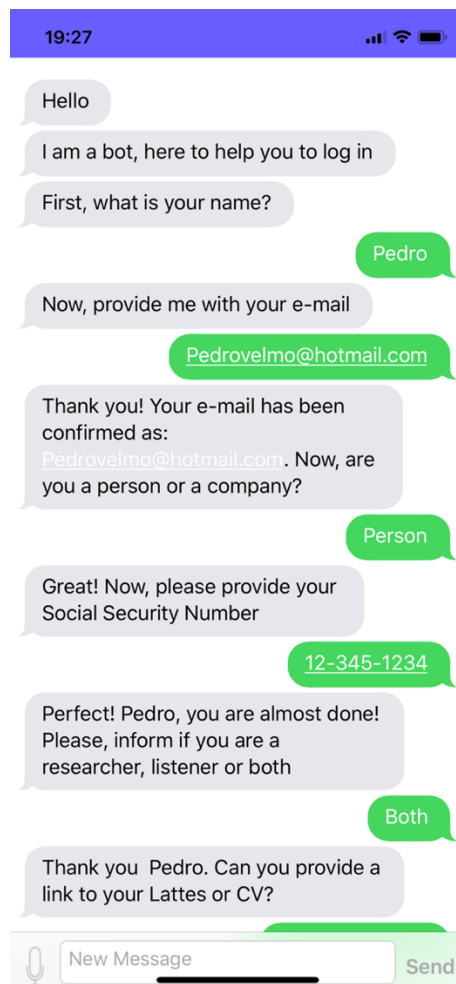


Figure 20: Login Chatbot for Person

5.4 Clarification Chatbot

Another proof of concept chatbot is the clarification chatbot: if the user wants to know more about the subject of an event, he can ask the chatbot to explain it. The chatbot can give a more detailed and longer explanation if the user is experienced in the subject, or a simpler explanation, with analogies, to beginner users (the user profile can be inferred through the use of the app).

The Information State for the clarification chatbot is defined as follows:

Non-Domain Related

- Utterance_List;

- DFResponse;
- Intent_List;
- Next_Utterance.

Domain Related

- Subject.

The non-domain related slots are the same as in the login chatbot. The slot *subject* stores the subject to be explained to the user.

Similarly to the previous chatbot, the Update Rule Agents are defined according to what slot they are monitoring. Therefore, the only URAs in this chatbot are the *SubjectAgent*, to monitor if the intent of the user is “*explainSubject*” and what subject the user wants explained, and the *InputAgent*, which is the same as in the previous chatbot. There is also an agent in the Task Agent Society, called *ExplainAgent*. This agent, called as a side condition of the *SubjectAgent*, receives a message from it containing the subject that the user wants explained and gets information about the subject from a database. This information is then returned to the *SubjectAgent*. Figure 21 shows these agents.

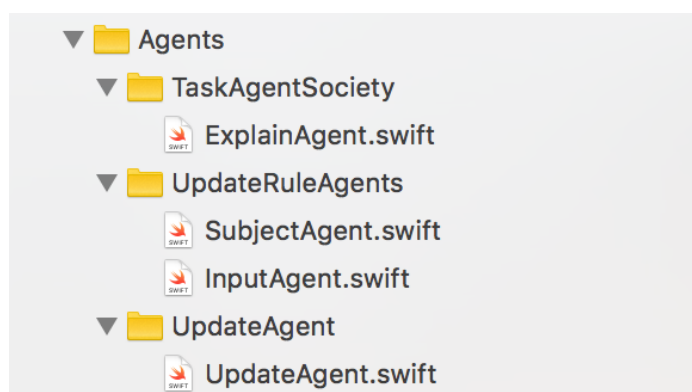


Figure 21: Agents in the Clarification Chatbot

Figure 22 shows Dialogflow’s interface for this chatbot, with the respective intents, and Figure 23 details the custom entities.

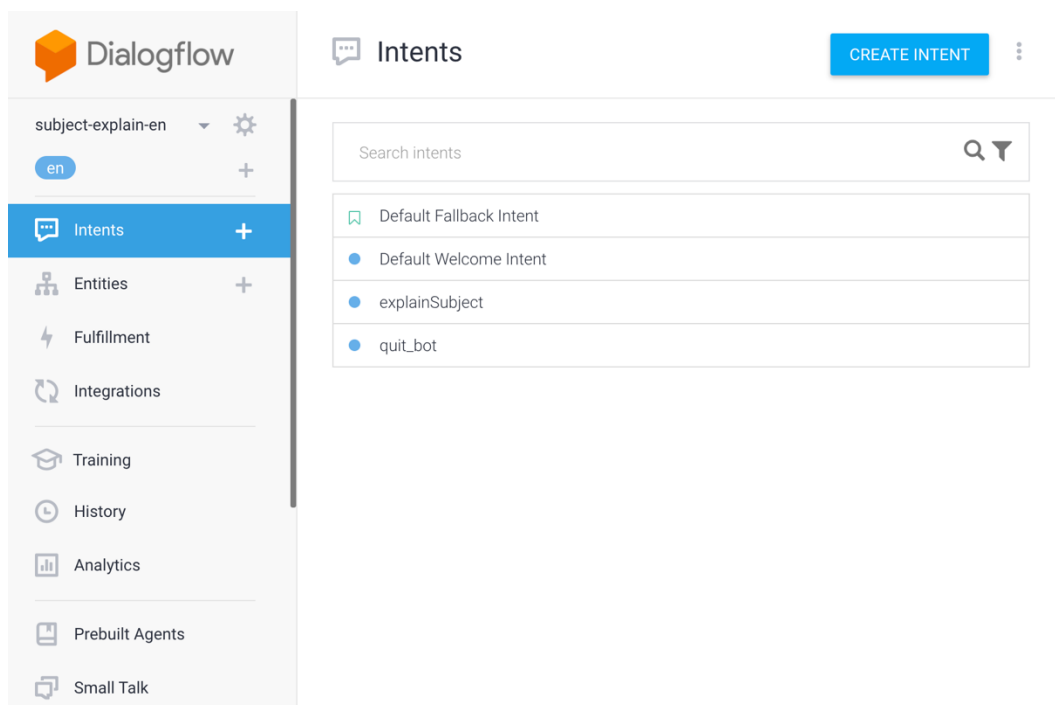


Figure 22: Intents in the Clarification Chatbot

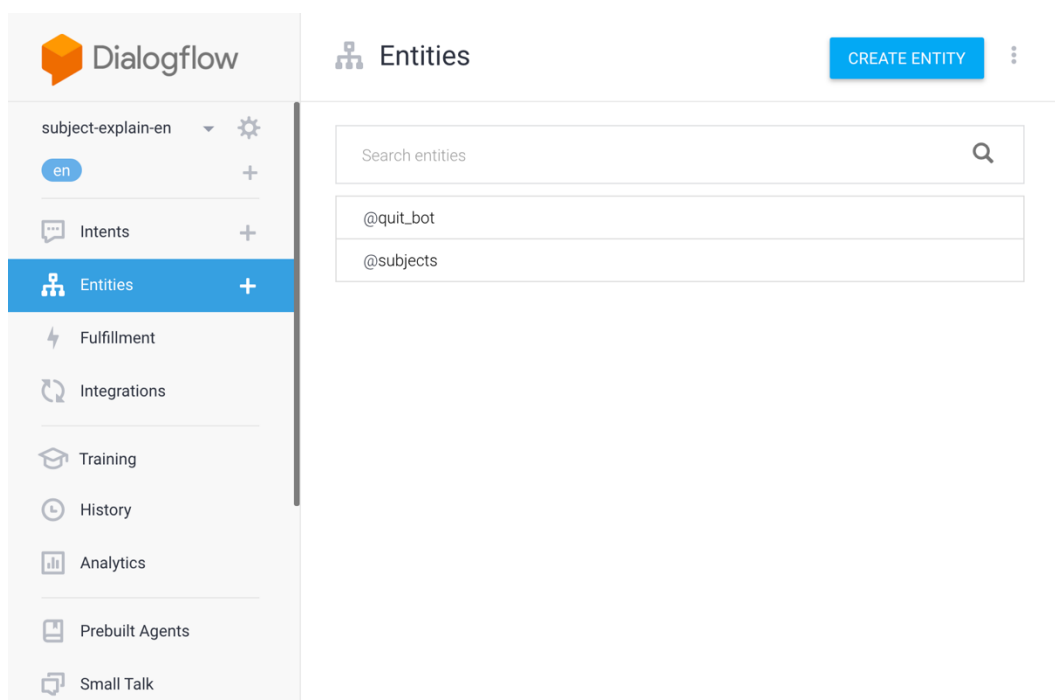


Figure 23: Custom Entities in the Clarification Chatbot

Figure 24 shows the dialogue in which a beginner user wants to know more about the subject of astronomy – in this case, the explanation is longer and more detailed.

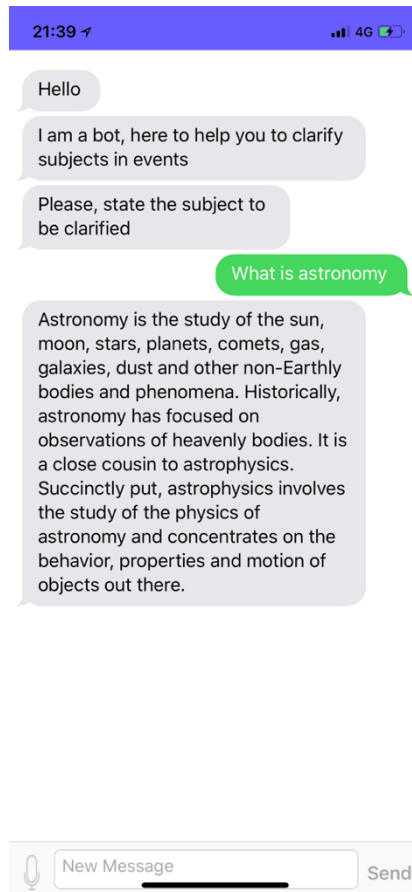


Figure 24: Clarification Chatbot for Beginner User

Figure 25 shows the dialogue in which an experienced user wants to know more about the subject of astronomy — in this case, the explanation is smaller and less detailed. You can see more of the dialogue in the following link: <https://youtu.be/m46oCJTqHLs>.

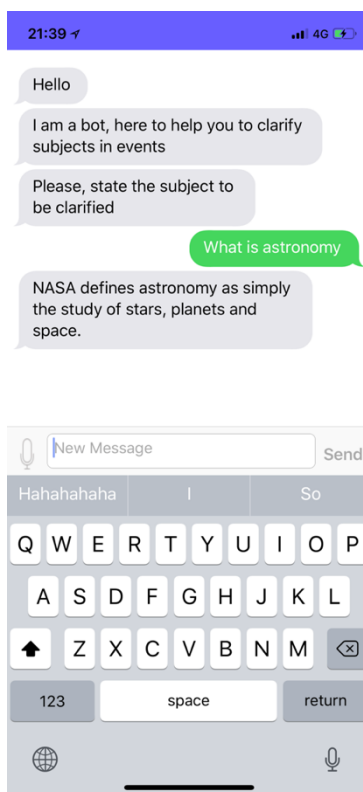


Figure 25: Clarification Chatbot for Experienced User

5.5 Event Chatbot

The event chatbot has been implemented to help users find events according to their interests and confirm presence in them.

The Information State for the event chatbot is defined as follows:

Non-Domain Related

- Utterance_List;
- DFResponse;
- Intent_List;
- Next_Utterance.

Domain Related

- Event_list.

The non-domain related slots are the same as in the previous chatbots. The slot *Event_List* stores the list of events that satisfy the user's queries.

Similarly to the previous chatbots, the Update Rule Agents are defined according to what slot they are monitoring. So, this application has the URAs

FindEventAgent, to find an event according to the user's queries, and *SelectEventAgent*, to select an event, besides the *InputAgent*. There is also an agent in the Task Agent Society, called *GetEventAgent*. This agent receives a message of the *FindEventAgent* containing the subject of the event that the user wants to attend and its start and end dates, and gets the list of events which satisfy those conditions in a database. This information is then returned to the *FindEventAgent*. Figure 26 shows these agents.

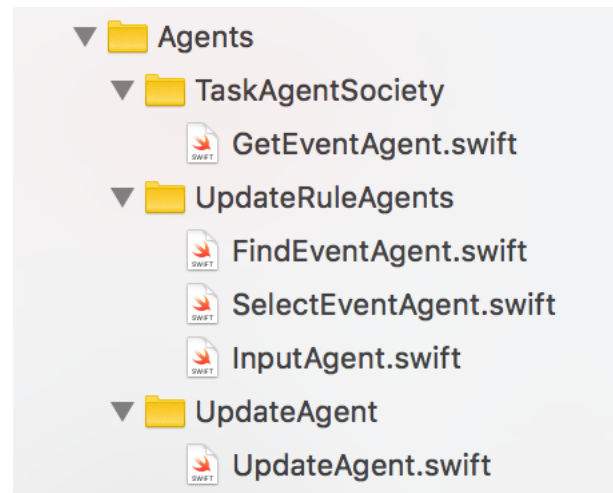


Figure 26: Agents in the Event Chatbot

Figure 27 shows Dialogflow's interface for this chatbot, with the respective intents, and Figure 28 details the custom entities.

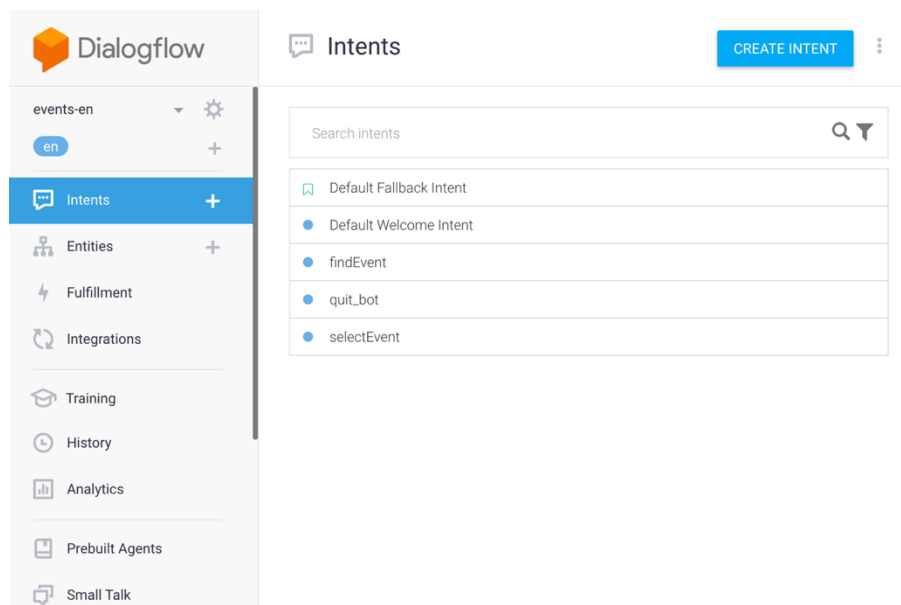


Figure 27: Intents in the Event Chatbot

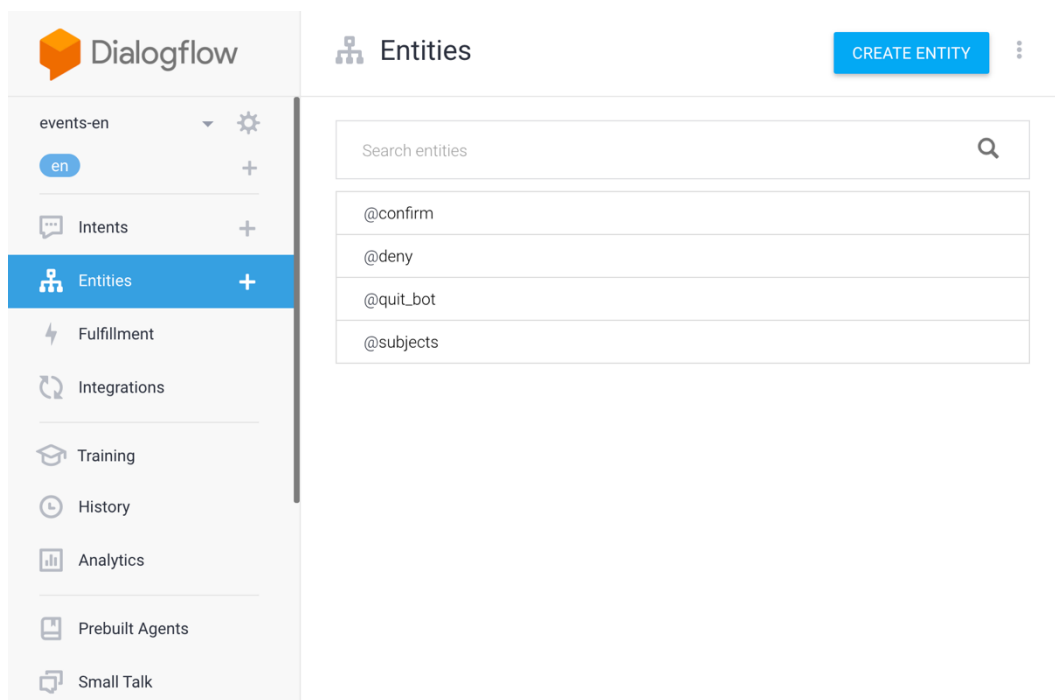


Figure 28: Custom Entities in the Event Chatbot

Figure 29 shows the dialogue in the event chatbot. You can see more in the following link: <https://youtu.be/VLllo3FneIE>.

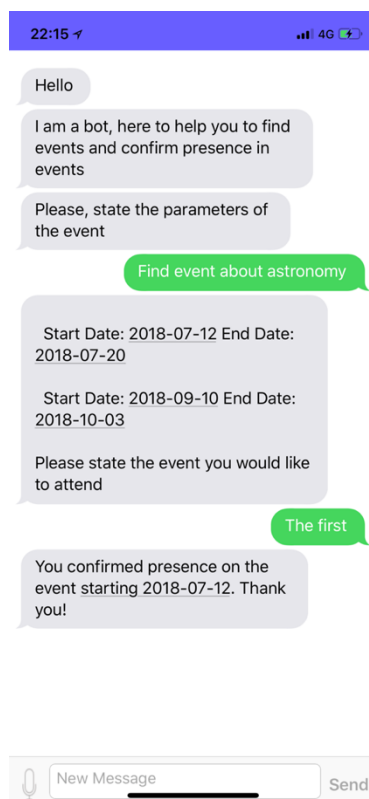


Figure 29: Event Chatbot Dialogue

5.6 Comprised Chatbot

Lastly, a chatbot comprising all the previous functions has been implemented. The Information State for the event chatbot is defined as follows:

Non-Domain Related

- Utterance_List;
- DFResponse;
- Intent_List;
- Next_Utterance.

Domain Related

- Event_list;
- Subject;
- Name;
- Email;
- Id;
- Interests;
- cvLink;
- User_Type;
- Person_Type;
- chatbotFunction.

The non-domain related slots are the same as in the previous chatbots. The domain related slots comprise the Information State from the previous chatbots, plus a slot called *chatbotFunction* to store the chatbot's current function: login, explanation or search for events.

In the same vein, the Update Rule Agents and Task Agents are a compilation of the agents in the previous chatbots, plus an Update Rule Agent called *FunctionAgent*, regarding the *chatbotFunction* slot. Figure 30 shows these agents.

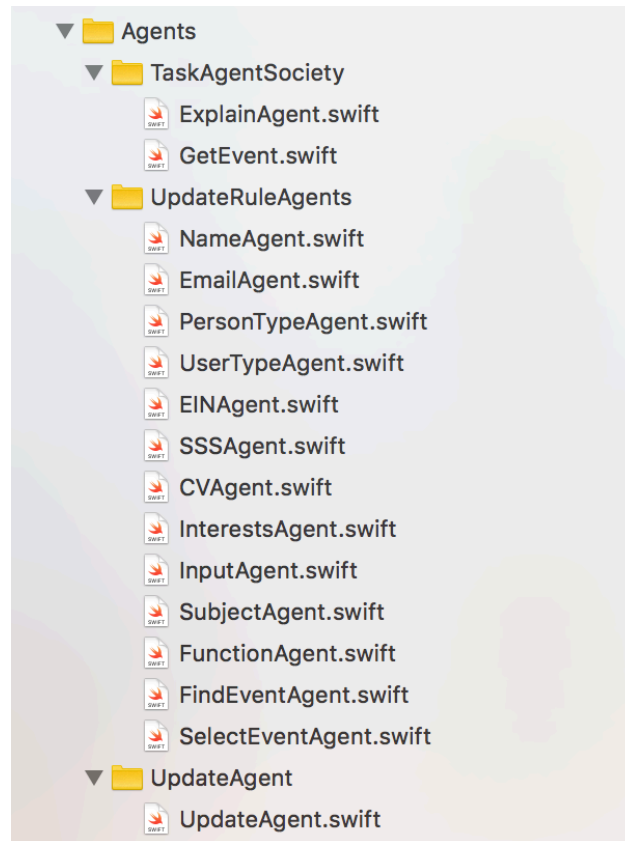


Figure 30: Agents in the Comprised Chatbot

Figure 31 shows Dialogflow's interface for this chatbot, with the respective intents – comprising all the intents of the previous chatbots, plus one intent to recognize the chatbot function. Figure 32 details the custom entities – similarly to the intents, they comprise the entities of the previous chatbots with an entity to identify the chatbot function.

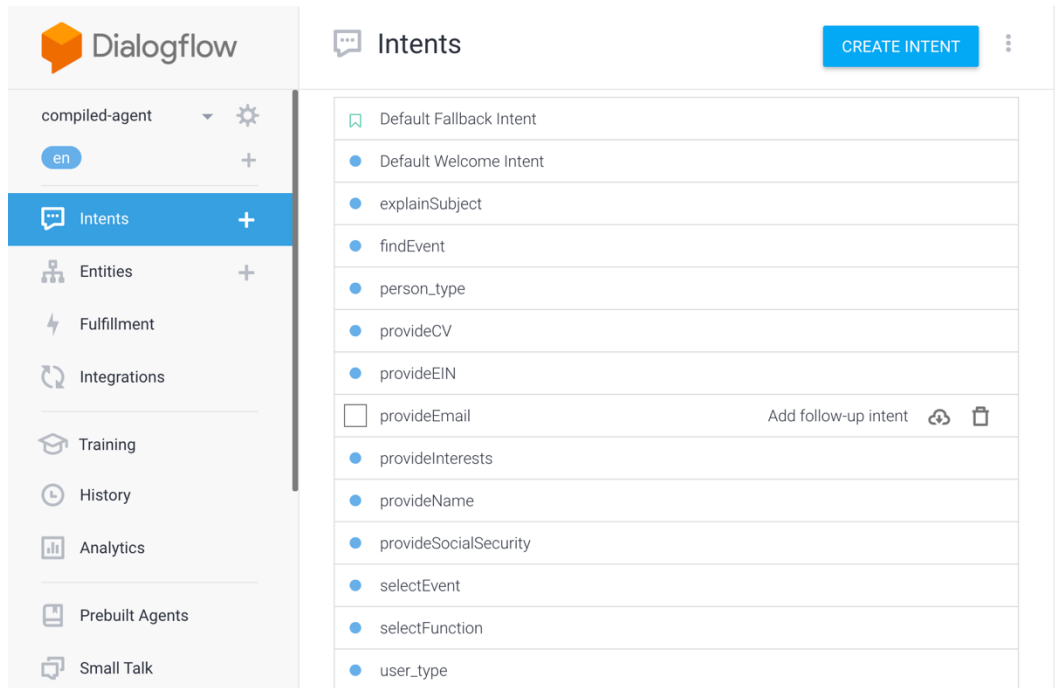


Figure 31: Intents in the Comprised Chatbot

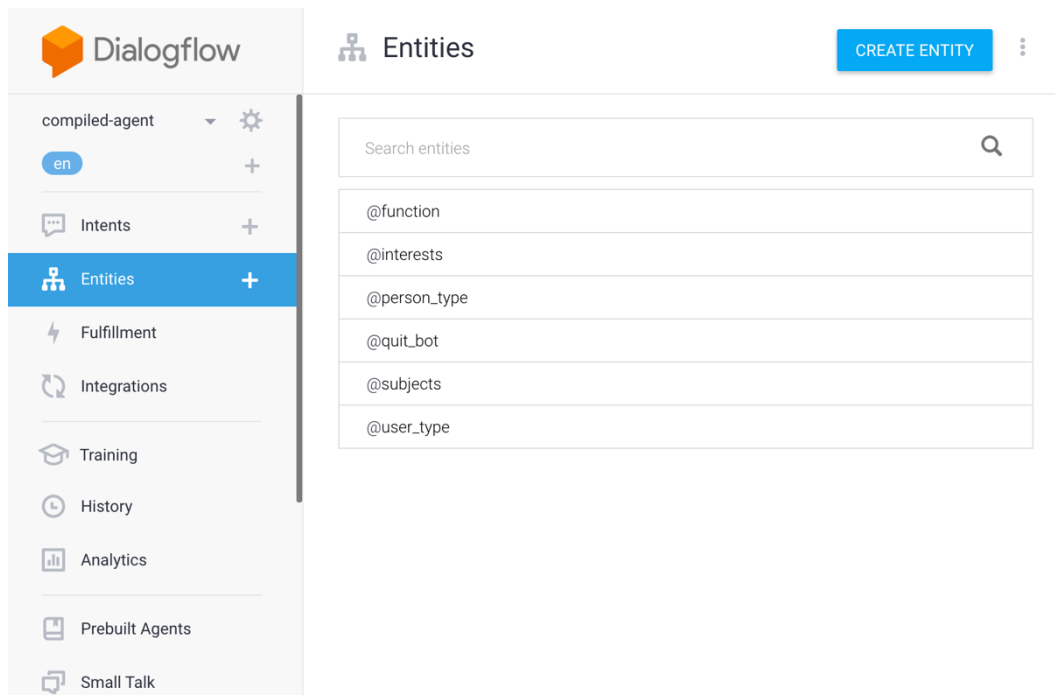


Figure 32: Custom Entities in the Comprised Chatbot

Figure 33 shows the dialogue in the comprised chatbot. You can see more in the following link: <https://youtu.be/uGUbIB3EIXY>.



Figure 33: Comprised Chatbot Dialogue

5.7 Discussion

This section answers the research questions **#RQs6**: “Does the proposed solution allow for the development of chatbots for different tasks?” and **#RQs7**: “Does the proposed solution allow dialogue control in the application?”.

The proof of concept chatbots, created for the app being developed by LES and IDOR, extend the iBot framework and distribute domain tasks to agents. By doing so, the chatbot is able to fulfill its goals. Therefore, developers using iBot are able to create chatbots in different domains.

The use of the Information State approach, underlying ADMP, also provides for a robust dialogue control in the application, since the Information State contains the necessary information to be collected by the system. The turns in the dialogue are controlled by updates in the Information State, allowing developers to create a dialogue flow according to what information has been provided by the user during the dialogue. Moreover, the Information State Updates are executed by the Update

Agent. This Update Agent provides a meta-level to reason about heuristics, adding another layer of dialogue control to the application.

It is important to note that, during development, there was no need to create or implement new components besides the ones already present in the Core Architecture proposed for iBot. These components were extended, as described in the sub-chapters above, to instantiate chatbots using iBot.

In terms of the Natural Language Interpreter module, created with Dialogflow in the implementation, it is interesting to note that with a greater number of intents, it becomes harder to correctly recognize what is the underlying intention by the user. In fact, in the comprised chatbot specially — which has the biggest number of intents — the system identified the wrong intent at times. In this case, a greater number of utterances are needed in each intent to improve recognition by Dialogflow.

To create the software agents, the *iMobile* framework was used (MIRANDA & LUCENA, 2017). It provided an invaluable tool to integrate multi-agent technology with iOS, and since Dialogflow also provides an iOS SDK, it allowed the development of the entire application in XCode. It is important to note, however, that each agent in *iMobile* is a thread, and during development XCode crashed at times because there were a lot of different threads active at the same moment, and the IDE could not support them all. It is beyond the scope of this work to analyze if this is a limitation of the current version of XCode or *iMobile*, but it remains a limitation of the implementation nonetheless.

6 Conclusion and Future Work

This Chapter presents the conclusions of the dissertation with an analysis of the iBot framework's main contributions and results, as well as indicates future directions of work.

6.1 Main Contributions

This work presented iBot framework, which:

- (i) Allows developers to create chatbots in different domains;
- (ii) Maintains a robust dialogue control of the information in the dialogue of the application.

In order to achieve these objectives, iBot combines concepts and techniques of software agents and multi-agent systems, to perform tasks in the application's domain by distributing them to the Task Agent Society, and to perform dialogue management using the information state approach by using ADMP.

To validate this framework, 4 (four) proof of concept chatbots were developed using iBot, to be integrated in a mobile app being developed jointly by LES PUC-RJ and IDOR to help stimulate scientific divulgation. These chatbots handle different tasks and domains and by using the information state approach, they are able to maintain a robust dialogue control in the application, using the information obtained in the dialogue by the user to control the flow of the conversation.

6.2 Main Limitations

Sub-chapter 5.7 describes the limitations of the iOS implementation of iBot. Therefore, this sub-chapter will focus on the more general limitations of the iBot framework at the moment of writing. First, although it purported to provide a blueprint for developers to create chatbots in different domains that maintained dialogue control in the application, the lack of a defined set of requirements and best practices in iBot can provide a difficulty for using the Core Architecture

presented in Chapter 4, since there is a great number of techniques and solutions for each of its main components. For example, for Natural Language Interpretation developers can use Microsoft Azure, Google DialogFlow, IBM Watson, among others, and for creating software agents they can use Jason, JADE, iMobile, among others.

So, it may be difficult to use iBot as a blueprint for creating chatbots with so many development approaches available. While the Core Architecture of the framework was purposefully developed to be free of requisites in terms of programming languages and platforms, a set of best practices for the framework is essential. The next sub-chapter describes future work that will revolve around the development of these best practices.

Moreover, iBot has been conceived as a general framework for the creation of chatbots in different domains. While the initial validation with the user scenario showed that the solution is pointing in the right direction, it is hard to account for every need and eventuality in developing chatbots. Therefore, the constant use and monitoring of the framework, as well as user feedback, will be used in order to keep improving the solution.

6.3 Future Work

Future work will revolve in two directions. First, the proof of concept chatbots will be improved according to specifications from IDOR in order to be integrated in the mobile app. Following the Chatbot Best Practices from IBM (CUMMINS, 2018) the conversational agents will be tested with users, since “supporting natural user interactions is the defining characteristic of the system”. Also, they will be continuously monitored and tuned according to user feedback.

The second direction involves the continuing development and evaluation of iBot. In order to develop best practices for the use of the framework, establishing a reusable foundation for developers, a comprehensive survey of development technologies for chatbots and for MAS will be performed. From the results of this survey, similar chatbots will be developed using more technologies and different combinations of integrations between cloud-based natural language applications and IDEs for developing MAS. Then, a comparison between these conversational agents will be performed, using the GQM paradigm (BASILI, 1992) (BASILI &

ROMBACH, 1988) and software metrics, such as the Chidamber and Kemerer (CK) metrics (GARCIA, 2004). The results of this comparison will point to the best development approaches in using iBot.

7 References

ALVES, P.; VIANA, M.; LUCENA, C. **An Architecture for Autonomous Normative BDI Agents based on Personality Traits to Solve Normative Conflicts.** Proceedings of the 10th International Conference on Agents and Artificial Intelligence, 2018.

APPEL, Ana Paula et al. **Combining Textual Content and Structure to Improve Dialog Similarity.** arXiv preprint arXiv:1802.07117, 2018.

BASILI, Victor R. **Software modeling and measurement: the Goal/Question/Metric paradigm.** 1992.

BASILI, Victor R.; ROMBACH, H. Dieter. **The TAME project: Towards improvement-oriented software environments.** IEEE Transactions on software engineering, v. 14, n. 6, p. 758-773, 1988.

BASKAR, Jayalakshmi; LINDGREN, Helena. **Cognitive architecture of an agent for human-agent dialogues.** In: International Conference on Practical Applications of Agents and Multi-Agent Systems. Springer, Cham, 2014. p. 89-100.

BICKMORE, Timothy; GIORGINO, Toni. **Health dialog systems for patients and consumers.** Journal of biomedical informatics, v. 39, n. 5, p. 556-571, 2006.

BORDINI, Rafael H.; HÜBNER, Jomi Fred; WOOLDRIDGE, Michael. **Programming multi-agent systems in AgentSpeak using Jason.** John Wiley & Sons, 2007.

BUCKLEY, Mark; BENZMÜLLER, Christoph. **An Agent-based Platform for Dialogue Management.** Proceedings of the Tenth ESSLLI Student Session, Edinburgh, Scotland, p. 33-45, 2005.

BUCKLEY, Mark; BENZMÜLLER, Christoph. **An agent-based architecture for dialogue systems.** In: International Andrei Ershov Memorial Conference on Perspectives of System Informatics. Springer, Berlin, Heidelberg, 2006. p. 135-147.

CAHN, Jack. **CHATBOT: Architecture, design, and development.** University of Pennsylvania School of Engineering and Applied Science Department of Computer and Information Science, 2017.

- COHEN, Philip R. et al. **An open agent architecture**. In: AAAI Spring Symposium. 1994.
- CUMMINS, H. (2018). **Chatbot Best Practices - IBM Cloud Blog**. Retrieved April 30, 2018, from <https://www.ibm.com/blogs/bluemix/2018/01/chatbot-best-practices>
- GARCIA, A. F. (2004). **Objetos e agentes: uma abordagem orientada a aspectos**. Rio de Janeiro.
- CAVALIN, P.; GATTI DE BAYSER, M. (2017) **Tutorial on Architectures and Algorithms for Conversational Agents**. 16th International Conference on Autonomous Agents and Multi-agent Systems (AAMAS 2017).
- Dialogflow - Basics**. (2018). Retrieved April 27, 2018, from: <https://dialogflow.com/docs/getting-started/basics>
- Instituto D'Or de Pesquisa e Ensino | IDOR**. (2018). Retrieved April 11, 2018, from: <http://www.idor.org>
- Les - Home**. (2018). Retrieved April 11, 2018, from: <http://www.les.inf.puc-rio.br/>
- LEVINSON, S. (2016). **Chapter 9: Speech Acts**. In HUANG, Y. Oxford Handbook of Pragmatics.
- LIN, Bor-shen; WANG, Hsin-min; LEE, Lin-shan. **A distributed architecture for cooperative spoken dialogue agents with coherent dialogue state and history**. 1999.
- Pesquisa revela que brasileiro gosta de ciência, mas sabe pouco sobre ela | Unicamp**. (2018). Retrieved April 11, 2018, from: <https://www.unicamp.br/unicamp/ju/noticias/2017/09/25/pesquisa-revela-que-brasileiro-gosta-de-ciencia-mas-sabe-pouco-sobre-ela>
- RAO, Anand S. et al. **BDI agents: from theory to practice**. In: ICMAS. 1995. p. 312-319.
- RFC 439 - PARRY encounters the DOCTOR**. (2017). Retrieved 28 February 2018, from <https://tools.ietf.org/html/rfc439>
- SERBAN, Iulian Vlad et al. **A survey of available corpora for building data-driven dialogue systems**. arXiv preprint arXiv:1512.05742, 2015.
- SHIEBER, Stuart M. **Lessons from a restricted Turing test**. arXiv preprint cmp-lg/9404002, 1994.
- SMITH, Barry. **John Searle: From speech acts to social reality**. John Searle, v. 1, p. 17-18, 2003.

E SOUZA, Pedro Augusto da Silva; MIRANDA, Andrew Diniz da Costa; DE LUCENA, Carlos José Pereira. **iMobile: A Framework to Implement Software Agents for the iOS Platform**. ICSEA 2017, p. 125, 2017.

TRAUM, David R. **Speech acts for dialogue agents**. In: Foundations of rational agency. Springer, Dordrecht, 1999. p. 169-201.

TRAUM, David R.; LARSSON, Staffan. **The information state approach to dialogue management**. In: Current and new directions in discourse and dialogue. Springer, Dordrecht, 2003. p. 325-353.

TURING, Alan M. **Computing machinery and intelligence**. In: Parsing the Turing Test. Springer, Dordrecht, 2009. p. 23-65.

WALLIS, Peter. **Believable conversational agents: Introducing the intention map**. In: International conference on autonomous agents and multiagent systems. 2005. p. 17-22.

WEINBERGER, M. **Why Amazon's Echo is totally dominating — and what Google, Microsoft, and Apple have to do to catch up**. (2017). Retrieved February 28, 2018, from <https://finance.yahoo.com/news/why-amazons-echo-totally-dominating-133000032.html>

WEIZENBAUM, Joseph. **Computer power and human reason: From judgment to calculation**. 1976.

WHITE, C. (2018). **The Chatbot Never Sleeps: How We Created a Chatbot Integration with Mendix That Enables 24/7 Customer Service - Watson**. (2017). Retrieved April 30, 2018, from: <https://www.ibm.com/blogs/watson/2017/08/chatbot-integration-with-watson-and-mendix-enables-24x7-customer-service/>

WOBCKE, Wayne et al. **A BDI agent architecture for dialogue modelling and coordination in a smart personal assistant**. In: Proceedings of the 2005 NICTA-HCSNet Multimodal User Interaction Workshop-Volume 57. Australian Computer Society, Inc., 2006. p. 61-66.

WOUDENBERG, Aswin van. **A Chatbot Dialogue Manager-Chatbots and Dialogue Systems: A Hybrid Approach**. 2014. Dissertação de Mestrado. Open Universiteit Nederland.

Xcode - Apple Developer. (2018). Retrieved April 30, 2018, from: <https://developer.apple.com/xcode>