Heriot-Watt University

Master's Thesis

Combining task-based dialogue systems and chatbot technology for socially intelligent multimodal Human-Robot Interaction

Author:
Ioannis papaioannou

Supervisor:
Prof. Oliver Lemon

A thesis submitted in fulfilment of the requirements for the degree of MSc.

in the

School of Mathematical and Computer Sciences

August 2016

Declaration of Authorship

I, Ioannis PAPAIOANNOU, declare that this thesis titled, 'Combining task-based dialogue systems and chatbot technology for socially intelligent multimodal Human-Robot Interaction' and the work presented in it is my own. I confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: Ioannis Papaioannou

Date: 17/08/2016

Abstract

Nowadays, humans use voice interactive artificial systems to perform standard day-to-day tasks almost daily. Either to book a flight, retrieve information or ask about the latest news, it is an undoubted fact that voice interaction has brought human-robot interaction a huge step further. Today, task-oriented software can perform tasks with increasing complexity, to the point of having lengthy conversations with their users. Still, users rarely use this technology for more than a couple of minutes, or until the task handed to the system is performed.

In this master's thesis, a hybrid system that combines technology first introduced in the 60's that provide a simplified voice interaction with users using Natural Language (chatbots), with task-oriented systems that are used to perform more complicated tasks (dialogue systems) is introduced and implemented. The system is trained using Reinforcement Learning (a Markov Decision Process model) on the best action to take in each subsequent conversation turns.

The system is then evaluated against a version using a rule-based decision model, without the use of Reinfocement Learning.

Acknowledgements

I would like to thank my supervisor Prof. Oliver Lemon and Dr. Arash Eshghi, which without their help and guidance this project would not be possible. Special acknowledgments also go to Dr. Simon Keizer, for his invaluable assistance on many steps throughout my research.

Contents

List of Figures

List of Tables

Abbreviations

AI	**A**rtificial **I**ntelligence
AIML	**A**rtificial **I**ntelligence **M**arkup **L**anguage
ASR	**A**utomatic **S**peech **R**ecognition
CA	**C**onversational **A**gent
DL	**D**eep **L**earning
DM	**D**ialogue **M**anager
DMML	**D**ata **M**ining **M**achine **L**earning
DS	**D**ialogue **S**ystem
DV	**D**ependent **V**ariable
ECA	**E**mbodied **C**onversational **A**gent
HRI	**H**uman **R**obot **I**nteraction
IV	**I**ndependent **V**ariable
LG	**L**anguage **G**enerator
MDP	**M**arkov **D**ecision **P**rocess
NLG	**N**atural **L**anguage **G**enerator
NLU	**N**atural **L**anguage **U**nderstanding
POMDP	**P**artially **O**bservable **M**arkov **D**ecision **P**rocess
OO-MDP	**O**bject **O**riented **M**arkov **D**ecision **P**rocess
RL	**R**einforcement **L**earning
SDS	**S**poken **D**ialogue **S**ystem
TTS	**T**ext **T**o **S**peech

Chapter 1

Introduction

1.1 Motivation

According to Pieraccini [2005], most modern interactive speech applications, can either focus more on usability, usefuleness and task completion, or on naturalness and freedom of communication, according to whether they are designed for industrial or academic use. Also, Waibel [2004] argues that commercial spoken dialogue systems can have a prohibitive development cost. These notions boil down to some common problems of today's interactive dialogue systems:

- They are expensive to develop
- They don't feel natural enough
- They are not useful enough

These systems that are able to receive and respond to users using natural language, referred to as Conversational Agents (CA), mainly fall under two categories:

High-level goal-oriented dialogue systems such as complex systems that are using Deep Learning and other AI techniques or that possess speech recognition and temporal reasoning. One of the longest running research project in this field is TRAINS [Ferguson et al., 1996, Traum, 1996] and its successor TRIPS (The Rochester Interactive Planning System) [Ferguson, 1999]

These systems try to parse the user input into meaningful sentences within the specified domain[1], often retaining a memory of the conversation along the states and along user sessions.

They are mainly *task-based*, in an open or closed domain, and as they can be immensely complex, they tend to be used in more narrow context. Some well-known task-based CAs are Apple's SIRI and Google's Google Now. Although these two examples are widely used, and perform some tasks adequately, they cannot keep the user engaged for long, as they lack the natural conversation flow of other systems.

Low-level dialogue systems such as chatbots, *"seek to mimic conversation rather than understand it. These systems employ simple algorithms to return dialog with minimal or no maintenance of state and thus no consideration of context"* [Schumaker et al., 2007].

Although most these type of systems are very easy to implement and can be maintained easily, they have a very limited usage (e.g., only for entertaining purposes, or simple information retrieval). Promising research has also been made in the field, using a Neural Computational model to substitute the hand-crafter rules of a chatbot [Vinyals and Le, 2015].

1.2 Objectives

In this MSc project, the objective is twofold. Primarily, a task-based conversational agent is extended with features present in low-level chat-based dialogue systems. In chapter 2.2 a more in-depth analysis of *embodied conversational agents* is made. Secondly, the system will decide on which action to take on each consequent dialogue turns, based on the user utterances and multi-modal information (the user's distance from the agent), by using Reinforcement Learning, specifically a trained *Markov Decision Process* (MDP) model. The resulting proposed system will be able to switch between these two modes (the two modes being either the *chatbot* is issuing the response or the *dialogue system*) and be able to decide the next action to take, providing a novel system that is hopefully able to engage the users for longer periods of time, making it perceived as

[1] The term domain is explained in more detail in section 1.2

more than a mere task fulfilling machine. To evaluate this, a secondary system will be created, substituting the MDP module with a simple rule-based one, as described in Chapter 5.

The selected environment in which the system is designed to operate, is a shopping mall. The system could be installed within a stationary human-like robotic head like *FurHat*, placed in a strategic and central place within the mall, in order to interact with the shopping mall's users (visitors, staff, etc). This project though uses a simulated avatar (henceforward a *virtual agent*), instead of the actual robotic head.

As in the selected domain, the system has to communicate with different users and converse over different subjects, the amount of different utterances within these dialogues is huge. All these different points within a dialogue are described as *states*. The implemented system is able to distinguish whether a user just requires a specific, task related response (e.g. information request), or a more generic conversation can be initiated (e.g. the user is curious and stands to examine the agent more carefully, in which case, the system will take initiative and begin chatting with the user using chatbot mode). These features have been implemented by analyzing multimodal information, such as the user's speech, combined with the user's distance from the agent.

Summarizing, the objectives of this project are the following:

1. Combine a chatbot with a task-based dialogue manager to create a hybrid spoken dialogue system and enrich it with multimodal input
2. Create an MDP model in order to train the hybrid system using Reinforcement Learning
3. Evaluate this system against a baseline rule-based version of system instead of the MDP

1.3 Expected results and beneficiaries

This hybrid approach would result in an embodied system with shared features from both technologies, that would have the following properties:

1. Easy to maintain and develop
2. More engaging to the user by combining task-oriented features with entertaining ones
3. Able to decide the best action based on multimodal input.

Such a system could have a lot of applications where a conversational agent is required to keep the user engaged for longer periods of time, like shopping malls, conventions, Personal Assistants, automated telephone assistants, games, etc.

Chapter 2

Literature Review

2.1 The Imitation Game

Alan M. Turing, the father of modern computing, set a crucial question back in the early 50's, "Can machines think?". Although in his paper [TURING, 1950] extends to great lengths this notion and the ambiguity of it, he devised a test that he thought would assess whether a machine can think or not. Up to this day, there is a lot of controversy about whether Turing's Test is an actual proof of intelligence. [Shieber, 2006], [Block, 1981]

The purpose of the test was for a human evaluator to assess conversations between a human and a machine that is programmed to generate Natural Language conversation. The evaluator would not know which is who, and the conversation would be conducted using text only means. If the evaluator could not distinguish if the conversation was either held with a human or with a machine, then that machine would have passed the Turing Test.

2.2 Embodied conversational agents

An Embodied Conversational Agent (ECA) is an agent that is capable of interacting with the users using not only Natural Language, but also human-like features(e.g. gestures, facial expressions, etc) and/or bodyparts. These ECAs can be grouped into two

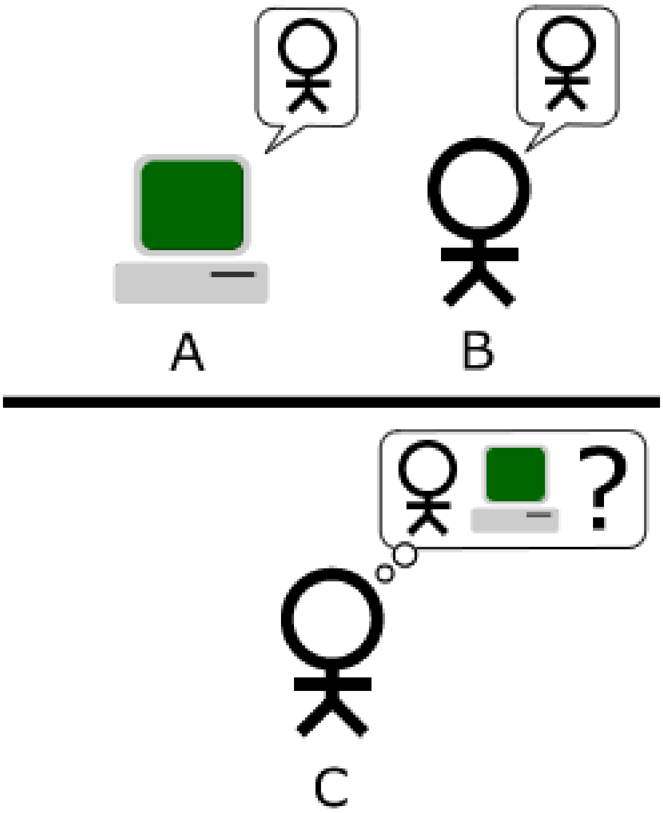
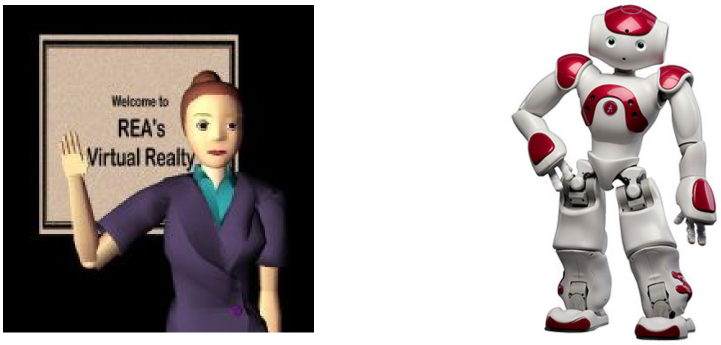


FIGURE 2.1: The Turing Test

main categories:

- **Virtual embodied CAs**, where the CA is a computer generated model of a humanoid.
- **Humanoid Robotic agent**, where the CA is running on a human-like robot.



(A) The REA conversational agent

(B) The NAO robot

The virtual embodied CA is usually used in simulations, and it may consist of only limited parts of the human body (e.g. head only, or face only), or it may use an entire human-like body to extend its interaction, as shown in Figure 2.2a. These types of CAs although easy to create most of the times, suffer from the *Mona Lisa effect* [Sato and Hosokawa, 2012], making it hard for the user to understand what the CA is actually gazing at. This gives a much less natural feeling in the *Human-Robot Interaction* (HRI).

The humanoid robot CAs on the other hand, do not suffer this effect. Due to the fact that they are a lot more complex and difficult in production, they don't always succeed in imitating the human gestures and movements as fluently as their counterparts though.

A lot of research has been done regarding the anthropomorphic features of a robot, and the effects they have on humans, such as the *uncanny valley* effect, which is a concept first identified by *Masahiro Mori* [Mori, 2010]. It is based on the idea that as the appearance of a robot comes closer to resembling a human being, the emotional response of the viewer also grows. Uncanny valley is the point of which the human imitation starts to become repulsive rather than connective for the user.

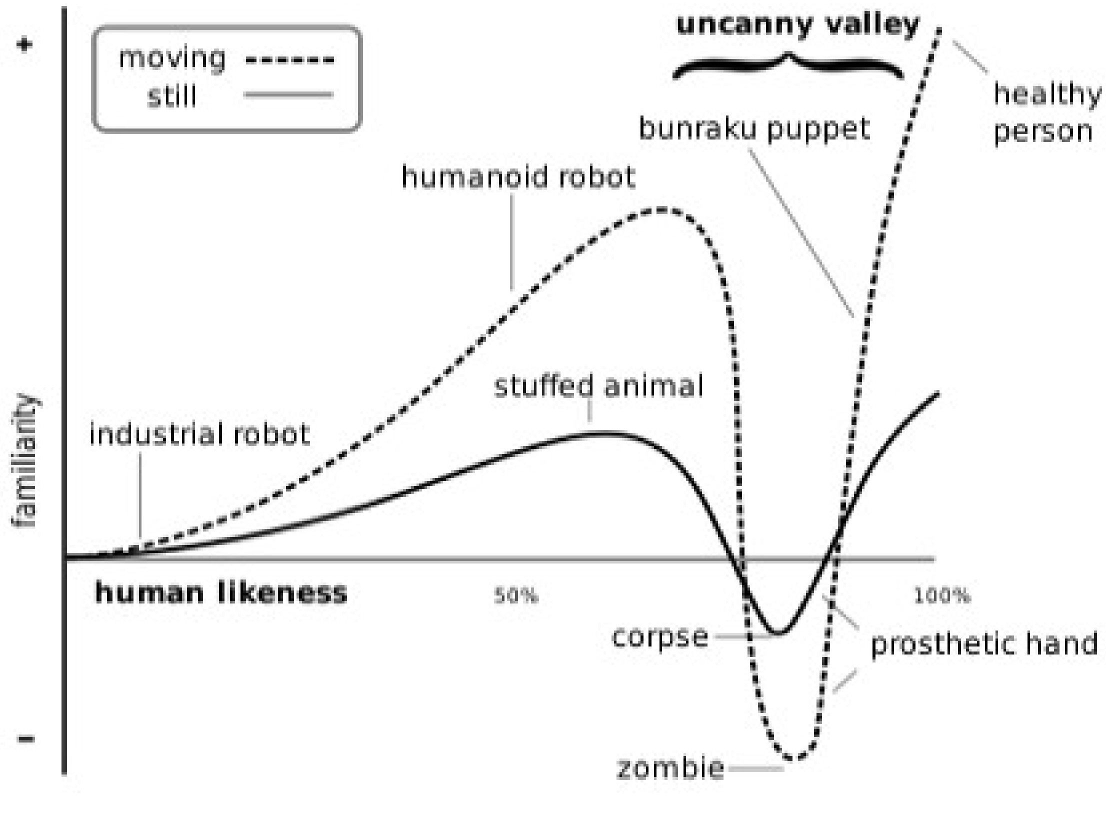


FIGURE 2.3: The uncanny valley

2.3 Chatbots

The chatbot (also known as chatterbot) technology was first introduced in 1960, in order to cover the needs of humans “to express their interest, wishes, or queries directly and naturally, by speaking, typing, and pointing” [Zadrozny et al., 2000].

The chatbots interact with the user via Natural Language, usually within a certain domain. The first chatbot system created was ELIZA by Joseph Weizenbaum, as early as 1960’s. The ELIZA system was mainly created for entertaining purposes, by searching simple keywords from the user input and applying specific rules that transformed the input into an appropriate output. Although the responses to the user might seem relevant and the conversation could be perceived as fluent, the ELIZA chatbot was merely using simple word substitutions, and in most cases just inverting the user’s question to a seemingly correct response, like:

- USER: ”I am feeling sick today”
- ELIZA: ”I am sorry that you feel sick today”

The significance of the ELIZA system was huge at the time, as it was the first program that appeared to pass the Turing Test [Weizenbaum, 1966]

It didn’t take long after that for the natural language research to grow fast, developing several chatbot systems like *”MegaHAL (Hutchens, 1996), CONVERSE (Batacharia et al., 1999), ELIZABETH (Abu Shawar and Atwell, 2002), HEXBOT (2004) and ALICE (2007).”* [Shawar and Atwell, 2009]

As the *Data Mining and Machine Learning* (DMML) techniques have improved, along with the introduction of corpora, processing tools like XML, etc., so did the chatbot systems, making them more appealing for commercial applications.

As stated in Chapter 1.1, although Chatbots perform well on certain occasions, their lack of actually parsing the user utterances creates several limitations. Chakrabarti and Luger [2015], provide a very common example of these limitations:

CHATBOT : Hello. How can I help you today?

USER : I am unable to log into my account. I forgot my password.

CHATBOT : Please click on the ”Forgot Password” link. You will be emailed a link that will help you change your password.

USER : I already tried that. That link takes me to a form to reset my password. But it requires me to enter my Customer Relationship Number. I do not have this number.

CHATBOT : The Customer Relationship Number can be found in the “Account Information” tab when you log in to your account.

USER : But I am unable to log into my account. I do not remember my password.

CHATBOT : Please click on the ”Forgot Password” link. You will be emailed a link that will help you change your password.

A closer analysis this conversation, reveals the underlying problem. Although the chatbot performs adequately in very short conversations, due to the fact that it acts only on the last user utterance, it can not handle longer conversations and the performance rate degrades rapidly.

2.3.1 A.L.I.C.E. and AIML

A.L.I.C.E. was developed in 1995 by Richard Wallace and won the Loebner Prize[1] in 2001, 2002 and 2004. Since then, its engine has been ported to multiple programming languages like Java, C/C++, PHP etc, and nowadays more than 200.000 chatbots have been developed based on ALICE [van Woudenberg, 2014]. ALICE uses the XML schema AIML (Artificial Intelligence Markup Language) a language designed specifically for creating stimulus-response chat robots [Wallace, 2009]. ALICE is using automatic pattern detection in dialogue data using supervised learning. [Wallace, 2009]

[1] The Loebner Prize is a competition on Artificial Intelligence, awarding the most human-like chatbot system

An AIML object consists of two basic units: *categories* and optional *topics*. Categories represent the knowledge the AIML object has and consist of the user input (referred as *pattern*) and the system's response (referred as *template*).[Shawar and Atwell, 2009]

2.3.1.1 Categories

Categories are knowledge objects of AIML. As stated before, the category consist of a *pattern* (user input) a *template* (system response) and an optional context.

```
<category>
        <pattern>I LIKE * </ pattern>
        <template>Nice! I also like <star/></ template>
</category>
```

In the example above, the user may say "I like basketball" where "basketball" will take the place of the *, and the AIML will use this knowledge to form its response using the pattern, in the style of "Nice! I also like basketball"

2.3.1.2 Recursion

AIML supports synonyms (recursion) using the tag <srai/>to match a different category to the utterance. In the example as shown in Wallace [2009],

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

the utterance "DO YOU KNOW WHO SOCRATES IS" will be recursively matched to the category "WHO IS SOCRATES". In the same manner, some utterances that can be phrased in different manners, such as a greeting, can be mapped to the same category.

```
<category>
        <pattern>HELLO</pattern>
        <template><srai>HI THERE</srai></template>
</category>
<category>
        <pattern>HEY *</pattern>
        <template><srai>HI THERE</srai></template>
</category>
<category>
```

```
        <pattern>HI</pattern>
        <template><srai>HI THERE</srai></template>
</category>
<category>
        <pattern>GREETINGS</pattern>
        <template><srai>HI THERE</srai></template>
</category>
```

2.3.1.3 Context

AIML can also deal with ellipsis (the omission of words from a sentence, while keeping the context unaffected) in the user input . For instance:

```
<category>
        <pattern>YES</pattern>
        <that>DO YOU LIKE MOVIES</that>
        <template>What is your favorite movie?</template>
</category>
```

The users response “YES” has a meaning only as an answer to the previous question of the system “Do you like movies?”. So in this example, when the system matches the user response “YES” to the system’s question, it can proceed to the next utterance “What is your favorite movie?”

2.3.1.4 Predicates

AIML supports predicates by using the <set>and <get>tags. In a *situated* dialogue (where the system is engaging multiple users at once), it stores each user as a unique client ID using these predicates. For instance <set name=“name”>John </set>stores the string representing the user’s name “John” under the predicate named “name”. Consequently, the activation of <get name=“name”>will return “John”. The predicates are frequently used to remember pronoun bindings. For example, the template

```
<template>
        <set name="he">Samuel Clemens</set> is Mark Twain.
</template>
```

results in “He is Mark Twain”, although “he” is remembered as “Samuel Clemens”. Conversation topic can also be changed using the <set name=“topic”>tag.The designer of the chatbot is also able to decide whether a predicate will return the context between the predicate tags or the predicate's name. E.g. , *“ <set name=“it”>Opera </set>returns “it”, but <set name=“likes”>Opera</set>returns “Opera”.”* [Wallace, 2009]

2.3.1.5 Person

By using the operator <person>, which is the equivalent of <person><star/></person>, AIML manages to substitute pronouns like *“I'm”* and *“your”* with *“you are”* and *“my”* respectively. For example:

```
<category>
        <pattern>YOU DO NOT * </pattern>
        <template>Why do you think I don't <person/>?</template>
</category>
```

can be translated as “YOU DO NOT CONVINCE ME THAT EASILY” with the response of “Why do you think I don't convince you that easily?'

2.3.1.6 Graphmaster

All the categories are stored in a tree graph, managed by the Graphmaster object, in order to perform pattern matching efficiently. The Graphmaster can be visualized as the file system, with the pathname being an AIML pattern path and the files at the end of the path are the templates.

Shawar and Atwell [2009] describe the usage of the Graphmaster using this representation. If the user's utterance begins with the word X, and the root folder in this tree structure contains every pattern and template available to the system, then by using depth first search techniques, the Graphmaster will perform the search as follows:

If the root folder contains a subfolder starting with “_”, then go into that folder and search all words suffixed X. In case the word is not found, go back a level and find a subfolder that starts with the word X and search inside of it for the tail of X. If again it is not found, go back one level and search for a subfolder starting with “*” and try searching inside for all remaining suffixes of the input word following X. If again no match is found then return to the parent directory and make X the head of the input again. If a match is found then stop the search and return the template that belongs to that category.

2.4 Spoken Dialogue Systems

Analyzing human-to-human spoken interaction can solve several of the previously noted problems and notions like:

- *Human dialogue structure*: What are the main components that make a dialogue, how do participants maintain context, etc.
- *Cues of engagement*: How does each participant know when to start talking?
- *Grounding*: When a certain statement or context is agreed on by all participants
- *Ambiguity*: When utterances can have multiple meanings.

These are just a few examples of things that happen within human-to-human dialogues which chatbot technology cannot solve. This gap can be filled using *Spoken Dialogue Systems* (SDS) or simply *Dialogue System* (DS), to *"provide a human centric interface for any user to access and manage information."* [Cheongjae et al., 2010]

This convention implies the mutual exchange of information between the CA and the user, as opposed to the chatbot systems. As stated before, dialogue systems in most cases refer to *task-oriented dialogue systems*, where a specific task needs to be fulfilled. In this project, the focus is mainly on a *task-based* dialogue system, with the ability to accomplish well-defined tasks (e.g. to search for a coffee shop within a shopping mall).

2.4.1 Dialogue System Architecture and Components

A (spoken) Dialogue System typically consists of 5 basic interconnected components. [McTear, 2002]

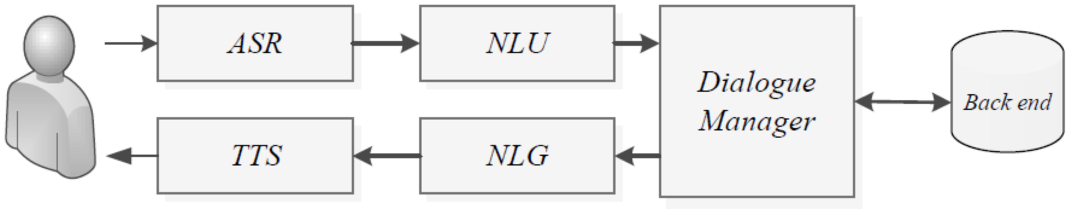


Figure 2.4: Typical Dialogue System components and workflow

The automatic speech recognizer (ASR) decodes acoustic signals into text strings. These strings are then interpreted by the language understanding (NLU) component. After the meaning of the utterance is found, the dialogue manager (DM) decides the action to be taken according to the meaning of the utterance. For instance, if the user requested directions to a certain place, the appropriate actions should be taken, but if he/she requested information on which shops have sales currently, then a database query might need to transpire, and a response will be given back to the user. The system's response is formulated by a natural language generator (NLG) and is eventually synthesized by the Text-To-Speech(TTS). Depending on the SDS architecture, a *back end* component might exist (e.g. a database to retrieve information).

2.4.1.1 Automatic Speech Recognition

The Automatic Speech Recognition transforms the analogue input signal (user speech) into digital, an then analyzes this signal for acoustic content. Examples of ASR are Google speech [Platform, 2016], Microsoft SAPI [API, 2016], Nuance Cloud [Nuance, 2016], etc.

The ASR ranks all the possible utterances generated by the audio signal, using a *confidence score* which *"reflects the reliability of correctness of the recogniser's output"* [Athanaselis et al., 2007]

2.4.1.2 Natural Language Understanding

Natural Language Understanding goal is to find the "meaning" of the user input, by parsing the input to a meaningful representation. A common technique used in most commercial applications is *slot-filling* parsing (also known as *frame-based approach*), where the dialogue system needs to get a specific set of information from the user before completing a task. This is widely applied for instance in information retrieval or booking systems.

2.4.1.3 Dialogue Manager

Cheongjae et al. [2010] describes the Dialogue Manager as *"the heart of the spoken dialog system because it coordinates the activity of all components, controls the dialog flow, and communicates with external applications."* The main role of the DM is to decide the next action the DS should take, based on the dialogue acts, user preferences and set of goals. For instance, in a fast-food ordering situation (using a slot-filling parsing) and given the following conversation:

SYSTEM : Hello. May I please take your order?

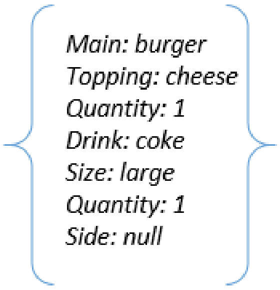
USER : Hi. I would like to have a cheeseburger and a large coke.

SYSTEM : Would you like some fries as well?

USER : No.

The system knows it requires some specific information before completing the order (slots) which are the main dish, the refreshment, the size, and the side order. The first input of the user filled all the required slots in one go, so the system required only the empty one (side order). By parsing then the user response *"no"*, it may fill this slot as *null* and process the order normally.

A possible representation of the final order could be like in the following frame:



Main: burger
Topping: cheese
Quantity: 1
Drink: coke
Size: large
Quantity: 1
Side: null

FIGURE 2.5: slot-filling frame

2.4.1.4 Natural Language Generator

Natural Language Generator is responsible for taking the appropriate information representation from a knowledge base and translating it into meaningful sentences.

The standard method in SDS is to use template-based generation in NLG, where the linguistic structure may contain gaps, producing well-structured results only when those gaps are filled [Deemter et al., 2005].

A simple example of a template-based system, formulated like the example in Reiter and Dale [1997], could be a semantic representation saying that the movie ”Star Wars” starts at 21:00 in screen room 4:

$$Movie(Title_{StarWars}, Time_{2100}, Room_4)$$

producing the associated template:

The movie [title] starts at [time] in screen room [room]

,

where the gaps (also known as *slots*) *title, time* and *room* are filled with information available in a lookup table.

2.4.1.5 Text-To-Speech generator

A Text to Speech generator (TTS) has the opposite functionality of the ASR. It takes the response generated by the NLG string and using a synthesizer outputs the response using a voice interface to the user.

2.5 Reinforcement Learning for decision making

In order for the Dialogue Manager to decide which action it should take on the next turn, two main approaches have been widely researched.

The first one is to hard code all the possible transition rules, according to knowledge the system has gained so far. This procedure can provide fast implementation in small systems and really restricted domains, the designer has full control over the states and transition and can be really simple using *if..then* rules. On the other hand, it can be a tedious procedure and is heavily prone to errors, while providing no guarantee of the optimality of the rules, since it is purely design dependent.

The second one uses Reinforcement Learning, specifically statistical modeling techniques [Levin et al., 2000], [Rieser and Lemon, 2011], usually by representing each possible dialogue state within a *Markov Decision Process* (MDP) or a *Partially Observable Markov Decision Process* (POMDP). Then the optimal action is decided by an *optimal policy*, trying to maximize its expected utility. This solution provides scalable, context-aware systems that provide natural responses. On the downside, the system needs to learn from experience, meaning it needs to be fed a lot of domain specific sample conversations. In this project, an *MDP* is used.

Reinforcement Learning (RL) is basically learning *"what to do–how to map situations to actions–so as to maximize a numerical reward signal"* [Sutton, 1998] when a learning agents is trying to achieve a specific goal while interacting with its environment. In Reinforcement Learning the agent is not told which action to take in each *state*[2], but instead is pushed to discover the best action in each state. This is done by a *reward* and *punishment* system, where the agents can either be only rewarded (when taking the correct actions), only punished if not, or use of both reward and punishment. The best action is selected by calculating the *accumulated discounted reward* at the end of the episodic interaction, as explained in the following subsections. It is worth stating that the possible states within a model (called *state space*) can either be finite or infinite, depending on the problem to solve. If the state space is infinite, then the task becomes a *continuous domain* [3] problem and a different approach is required. In this project, a finite state space is used.

In RL, usually there is a trade-off between *exploration* and *exploitation*, meaning that in order for the agent to maximize its accumulated reward, it should select already known actions that provide a big reward from previously trial-and-error (exploitation). But in order to know which actions are most effective in each state, it must first discover them by randomly picking actions (exploration). Setting these parameters is not a straightforward task for the designer, and is heavily depended on the problem to be solved. For instance in a stochastic task, each action should be picked several times before converging to reliable results [Sutton, 1998]. As a general notion, the agent must at first randomly pick an action and *progressively* favor those that yield the best result.

[2] *State* is the knowledge of the agent of its observable environment at a specific moment.

[3] A domain that has an infinite number of possible states, and is unlikely to visit the same state twice.

2.5.1 Markov Decision Processes

"A reinforcement learning task that satisfies the Markov property[4] *is called a Markov decision process, or MDP."* [Sutton, 1998] A Markov decision process (MDP) can represent the dialogue states as $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$, where $\mathcal{S}$ represents the state space, that all sets of possible states are, $\mathcal{A}$ all the possible actions, $\mathcal{T}$ the transition function for an action a from state s to reach state s', with a probability of $\mathcal{P}(s'|s, a)$, and $\mathcal{R}$ the reward gained if action a is taken in state s. *"The best action is defined as the action that maximizes the expected return for the agent, which corresponds to the expected long-term accumulation of rewards from the current state up to a given horizon".* [Rieser and Lemon, 2011]

MDPs consist of 4 basic subelements:

- A *policy*, that is basically a mapping from states to actions taken in those states. This is the outcome (or solution) of the MDP, where it can be as simple as a lookup table, whereas in other cases it may involve complex computation to get the best action in a specific state.
- A *reward function* that defines the goal of the problem. It basically informs the agent on how good a specific action is in a specific state., indicating the *immediate* desirability of a state. In a biological system comparison, a reward function would be the body providing pain or pleasure (punishment or reward).
- A *value function* on the other hand, indicates the *long-term* desirability of the states. This function calculates the value of each state, as the accumulated *expected* reward taking into account all the future states (and their rewards) that the agent is likely to visit, starting for that specific state. This allows the agent to select paths that although may yield a minimal immediate reward, the accumulated expected final reward is going to be better, since following that path will lead to states with a much higher reward. A human analogy would be the wisdom of a person when making farsighted goals (due to previous knowledge), even though the short-term outcome is not showing to be that promising.

[4] A *Markov property* dictates that the future states of a task or process are independent of the past states, and only depends on the current one.

- Finally, the *model* of the environment *"mimics the behaviour of the environment"* [Sutton, 1998], meaning that the model predicts the future reward and possible state transition, given a state-action pair.

The transition model is usually unknown so gathering the required training data from human-human interactions is impractical. That's why in most cases, a simulation is used to generate the needed interactions. In many cases a *Wizard-of-Oz* is used [Williams and Young, 2004],[Rieser and Lemon, 2008], which is actually a human-human interaction, where the "Wizard" is one of the humans using either TTS or voice modulation, so the other participant thinks he is actually talking to an artificial system. That way, only a few interactions are needed, and an optimal strategy can be found by extracting data from these interactions.

2.5.2 Partially Observable Markov Decision Process

Although this project uses a MDP approach, it is fairly easy to introduce a level of uncertainty in the domain, as referred to in section 7.2, Future Work. In a MDP solution, the dialogue state must be fully known, or *observable*. As this is often not the case, an extended framework of the MDP can be used, called *Partially Observable Markov Decision Process* (POMDP). In POMDPs, the possible solutions are notated as $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{O}, \mathcal{Z})$, whereas before $\mathcal{S}$ represents the state space, $\mathcal{A}$ the action space, $\mathcal{T}$ the transition probability and $\mathcal{R}$ the reward function. As the state now is not fully observable, $\mathcal{O}$ denotes the a set of possible observations by the system, according to its current knowledge, and $\mathcal{Z}$ is the probability $\mathcal{P}(o|s)$ of observing o in state s [Lison, 2015].

The system's knowledge at any given time is forming a *belief state*, which is the probability distribution over all possible states [Lison, 2015], which is constantly updated as the system's knowledge expands, using:

$$b'(s) = \mathcal{P}(s'|a, o) = \eta \mathcal{P}(o|s) \sum_{s \ni \mathcal{S}} \mathcal{P}(s'|s, a) b(s)$$

where b is the belief state, a is the action followed by observation o and η is a normalisation factor. The POMDP policy is then formulated by “mapping each possible belief state to its optimal action”[Lison, 2015].

Young et al. [2013] describes the observation probability function as a stochastic model M, while the decided action of each turn as the result of a second stochastic model P. On each turn of the dialogue, a reward is given based on a reward function R. During training, the dialogue model M and the policy model P are trying to maximize the accumulative sum of these small rewards.

2.5.3 Optimal Policy

A policy π is dictating the system action that can either be represented by mapping the states in a deterministic way to the actions, “or stochastically via a distribution over actions” [Young et al., 2013]. The *Bellman equation* (shown below) is used, in order to converge to an *optimal policy* that maximizes the discounted accumulated rewards gained by following that policy π in a state s (denoted as *value function* V^{π}),

$$V^*(s) = \max_a R(s,a) + \gamma \sum_{s'} P(s,a,s')V^*(s')]$$

where $V^*(s)$ represents the maximized V^{π} in a state s, R is the reward gained from performing action a in state s, P is the probability of taking action a in state s leading to state s', and $\gamma \sum_{s'} P(s,a,s')V^*(s')$ is the sum of probabilities of an action a within a state s of expected future values V^*, discounted by a factor γ. The discount factor γ is a number between 0 - 1 and represents how appealing immediate rewards are compared to long term ones. If the γ is set to 0, the immediate rewards are favored over long term ones, meaning that the agent will choose actions that provide the highest reward at any given time, irrespectively of any possible bigger rewards it could gather in the future, if it was to take an other action. In the same manner, a γ set to 1, means that there is no decay on the reward awarded at each state. This setting is crucial to learning algorithms, depending on the problem to be solved.

2.5.4 Q-Learning

As explained before, given a state space S and a possible action space in those state A, the agent has to learn the value of each action in those states. In Q-learning the *value* (remember that the value is the expected accumulated reward and the value of the next state discounted by a factor γ) of the *state-action* pair is called *q-value.* Initially those values are set to either an arbitrary fixed value, or set to random values depending on the design. Then the agent starts exploring the action-state space. After an action is taken in a state, an observation of the current environment is made, evaluating the outcome. If it leads to an unwanted outcome (meaning the agent got punished, or received no reward), the algorithm lowers the q-value of that action in that state, and so if the agent revisits that state, other actions with higher q-value might be selected. Following the same reasoning, if the agent is rewarded taking an action in a state, the q-value is increased, making the selection of that action more likely to take place the next time the agent is in that state. Then the q-value is updated following according to:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + a[r_{t+1} + \gamma \max_a Q(s_{s+1,a} - Q(s_t, a_t)]$$

Where $Q(s_t, a_t)$ is the old value (at the current time step), a is a learning rate factor, which controls how fast new values are overwriting the old ones (usually, we start at a very high learning rate, and progressively lower it near the end of the training) [Even-Dar, 2001], r_{t+1} is the expected *reward* in the next time step (according to previous knowledge), and $\max_a Q(s_{s+1,a})$ is the expected maximum *q-value* in the next time step.

It should be noted that in Q-Learning, the q-value is updated on the *previous* state-action pair, since the agent must actually try the action in order to evaluate it.

In order to select an action in Q-learning, a *policy* is followed. While Sutton [1998] agrees that the simplest way to select an action would be to simply select the action A_t with the highest q-value at each time step t, such as $Q_t(A_t) = max_a Q_t(a)$, exploiting the knowledge the agent has gained up to that moment, sometimes is sub-optimal, since it does not allow for much *exploration.* Instead, we can allow the agent to select the action with the best q-value *most of the time* (with probability $1 - \epsilon$), but also to have a small probability ϵ to select another applicable in that state *random* action with equal probability. This policy is called ϵ-*greedy*, and is also the selection policy followed in this project. Following this policy, allows the *exploitation* of past knowledge while at the same time allowing some *exploration* as well. In order to maximize the *exploration vs exploitation* output, usually the ϵ starts from a high value, and gradually decaying as the algorithm converges to the optimal q-values.

2.6 Related Work on hybrid systems

van Woudenberg [2014] researched and implemented a hybrid system, combining a Dialogue Manager with a Chatbot. The goal was to replace the normal substitution based model of the chatbot with a dialogue manager. Instead of parsing the user utterance into a meaningful semantic representation, it would get the requested goal of the user utterance and return a predefined response using Natural Language. It would try to clean up the user input by removing any unnecessary information from it, swap pronouns, correct spelling errors, etc., in order to break down the utterance into a much simpler representation of the user's goal.

The system implemented was able to communicate only via text, and was evaluated by 3 different subjects of different educational background. The system showed promising results on being capable of holding long conversations, but only in *issue-based* dialogues (meaning that *"it views dialogue as the raising and resolving of questions"* [van Woudenberg, 2014]).

Dingli and Scerri [2013] also proposed a hybrid system, merging a chatbot with a dialogue manager. Their proposed system would have access to both a local and external knowledge base, that along with the user dialogue input would be able to create a model of the user's intention and simultaneously keep track of the user's interactions with his world.

The chatbot module would use the *ChatScript* chatterbot engine, in order to normalize the user input into formal XML objects. This normalization is processed by modules called *"Action Modules"* handling various types of input.

The knowledge bases used are local (based on *Apache Jena*) and either external, online of offline (using *YAGO* and *ConceptNet*).

This prototype system was evaluated using actual users, conversing with the system *"following a predefined context and set of example tasks"* [Dingli and Scerri, 2013]. The system was successful achieving the objectives for creating a conversational agent with external knowledge base.

Both these system, although successful in providing a base semantic representation in the chatbot conversation, are somewhat lacking in terms of extensibility or using multimodal information to enrich the dialogue. The proposed in this thesis system, uses the chatbot mainly for longer engagement, while performing any task related actions according to the MDP trained policy, providing both the capability to use a wide variety of sensory input (as proposed in section 7.2), since Reinforcement Learning permits highly complex decision making, high task completion rate and engagement that goes beyond a typical task-based Spoken Dialogue System. In Table 2.1 a comparison between these systems and the one proposed in this thesis is shown.

Table 2.1: Comparison between some previous implemented systems

	RL	**Chat**	**Task-Based**	**Multi-modal**	**Evaluated**	**Implemented**
Proposed System	X	X	X	X	X	X
van Wouderberg (2014)		X			X	X
Dingli and Scerri (2013)		X	X		X	X

2.7 Conclusion

So far it is made clear that while the chatbot technology can be extremely effective when small dialogues with information exchange is taking place, like requesting for time/weather, train schedule, etc. They can not handle though more complex dialogues where actual knowledge of what exactly the user is saying is required . On the other hand, while this situation can be handled by a Dialogue System equipped with a Dialogue Manager, the design of such a system can be a tedious procedure, and communication often doesn't feel natural. Apart from this, Dialogue Manager systems tend to be more task-oriented than Chatbots, making the user wanting to engage the CA strictly only until the task is completed.

In this project, a novel system combining both technologies is implemented, in order not only to complete the tasks appointed to it, but also keep the user engaged for longer periods of time. Furthermore, the effectiveness of such a system after training it using an MDP model that the DM will use to decide each next action during the dialogue is investigated.

Chapter 3

Requirements Analysis

In order to achieve the goals set in Chapter 1, the system must cover the following requirements:

3.1 Mandatory

1. **Language:** The implemented system will speak and recognize the English language.

2. **Entertainment:** one of the main objectives of this project is to make conversational agents under certain domains more entertaining, in order to keep the user more engaged. This can be achieved by parsing the user input and understanding the context. If the context is appropriate, the chatbot mode is fired, retrieving relevant entertaining material from a local knowledge base, such as jokes, truth facts, etc.

3. **Be generic within the domain:** Although the implemented system of this project will fall under a shopping mall domain, the grammar must be generic enough on the input it can parse. E.g. it must be able to provide information given a product type, a specific shop/place within the mall, specific item etc. The system should be able to understand request for information and directions for at least 5 shops of 3 different types, and be able to provide the appropriate responses.

4. **Natural dialogue flow:** The dialogue flow with the user has to be as natural as possible, either the system being in the Chatbot mode or normal (Dialogue System) mode. *Rule-based template* (see section 2.4.1.4) is used for the NLG.

5. **Mixed initiative:** The system should be able to respond to user request, but also engage with and attract attention of users. The system should be able to initiate the dialogue as well (agent initiative) and not only respond to questions.

6. **Switching between modes:** Should be able to successfully switch between the *chatbot* mode –where it will be able to provide general information (about the time, shopping mall opening and closing times, etc) and engage in small-talk, and the *dialogue system* mode – where it will perform more complex task-based dialogue.

7. **Trainable System:** Be able to be trained according to the environment by learning through interaction, using Reinforcement Learning.

3.2 Optional

1. **Live internet feed:** Be able to deliver latest news.

2. **Humorous:** Be able to tell jokes on demand.

3. **Situated dialogues:** Where the agent is aware of its environment and that the participants share with. This allows multiple users to be attended at once.

4. **Response time:** The response time from the time the user finishes his utterance till the time the system starts speaking the response should be as short as possible.

Chapter 4

Legal, Ethical and Professional issues

As the system is designed to work within a shopping mall, measures should be taken towards appropriate responses according to the user. The main tasks the agent will need to complete will be mainly direction and information giving, and it is the designer's responsibility to make sure the system provides correct information.

As the chatterbot does not understand the user input's meaning, it may respond in a reprehensible or inappropriate way. Although this phenomenon is not always controlled by the designer, it is the designer's task to take measures in order to prevent this as much as possible (e.g. by checking the output for offensive or otherwise inappropriate responses).

During the evaluation, all subjects have signed an informed consent. Apart from that, all data collected during this experiment have been anonymized.

All tools used to implement the system are open source and fall under the GNU license, apart from the NuanceCloud API used for the semantic grammar (see chapter 5), which uses an evaluation license provided by Nuance Developers [Developer.nuance.com, 2016]. All the evaluation took place within the lab.

Chapter 5

Implementation

As explained in section 1.2 two versions of the system have been evaluated. The first, acting as a baseline, in which the actions are decided using handcrafted rules, and the other wherein the actions are learned through Reinforcement Learning. The tools needed to build those can be broken down to 3 individual but intertwined software modules:

- *Program AB* that will compile and run the chatbot module
- *BURLAP*, a RL framework to train the MDP policy
- *IrisTK*, acting as an integrator between all other subsystems, as well as handling *speech recognition* and *voice synthesis*.

5.1 BURLAP - Reinforcement Learning

Brown-UMBC Reinforcement Learning and Planning (BURLAP) [MacGlashan, 2016] is a java code library, for learning and planning algorithm development using a very flexible system to define any kind of domains used in Reinforcement Learning. The library also includes performance analysis and domain visualization tools. By using *"object-oriented Markov Decision process (OO-MDP) formalism, which represents states as a set of objects in the world"* [MacGlashan, 2016], BURLAP is able to express domain representation and task definition in an extremely flexible way. BURLAP supports many predefined domains and learning algorithms, but by usage of Java interfaces, the designer is able to implement his own domains, fit to use in any project.

BURLAP was selected for this project primarily due to the fact that it is well maintained, provides vast flexibility and abstraction in creating the appropriate RL models while being well structured as a framework, and since it is Java-based, makes the integration easier.

Since BURLAP uses OO-MDP (Object Oriented - MDP), state, reward function, termination function, etc, are actually Java class objects. This provides easy maintainability, robustness and easy reading. In order to implement the MDP, some basic methods have to be implemented. These are the *state representation*, the *actions*, the *reward function*, the *termination function* that indicates when the agent has reached the final state (In this experiment it is the end of the dialogue, when the user leaves the vicinity of the agent) and the *initial state.* These methods create the *domain* of the agent.

5.1.1 State Representation

In this project, *states* represent the agent's knowledge about its environment at any given time. Basically, its a simple reading of all the variables that matter to it, like the presence of a user, the distance he/she has to the agent, what is the last thing the user said, etc. States are represented as a simple Java Object Class, with each variable (called *attributes* in BURLAP) being a method or variable within the class. By following this logic, the states are defined by 13 (mostly boolean) attributes as follows:

- **MODE**: the current mode in which the agent is in (chat mode or task mode). Chat mode has a value of 0 while task mode a value of 1.
- **DISTANCE**: denotes the distance of the engaged user from the Kinect sensor (see section 5.2). Although the Kinect sensor returns a *float* value for distance, this value is binned in 3 categories: close (value of 0), medium (value of 1) and far (value of 2). This is done for simplification purposes, in order to avoid making the problem *continuous domain.*
- **USRENGCHAT**:"User Engage Chat" is a boolean attribute that flips to *true* when the user initiates chat (as described further this chapter).
- **TIMEOUT**: is another boolean attribute, where is *true* if the user has stayed silent for a set period of time (during the experiments it was set at 5"). This attribute is mainly used so the agent learns to try to reengage the user.

- **LOWCONF**: "Low Confident" turns to *true* if the recognized by the ASR user utterance has a confidence score[1] below 0.7.
- **TASKFILLED**: denoted whether the user has given a specific task to the agent.
- **TASKCOMPLETED**: this attribute turns to *true* if the agent has taken the appropriate actions to perform the task given by the user. It is primarily used during the training of the MDP model.
- **USRENGAGED**: "User Engaged" indicates whether a user is engaged by the agent or not. This attribute states the *initial state* and *termination function.*
- **CTXTASK**: This String attribute holds the task context. This value can be either *coffee*, if the user asks something related to coffee, *electronics*, *clothing*, or *directions.*
- **SGOODBYE**: checks if the user has said goodbye (boolean attribute).
- **USRTERMINATION**: "User Termination" becomes true when the user abruptly terminates the conversation. This denotes the end of the conversation (or *episode*) and is heavily punished as described further down this chapter.
- **TURNTAKING**: This boolean attribute marks whose turn is during the conversation. Value 0 means it is the agent's turn while value 1 means it is the user's.
- **PREVACT**: "Previous Action" is a special state attribute, that holds the *action id* of the action the agent took during its previous turn. This provides some form of context to the training simulation, on how the simulated user should react (as explained further down in the chapter).

creating a state space S of 98.304 states.

5.1.2 Action Space

The possible actions the agent can take, form the *action space.* The agent was able to take the 8 following actions in this domain:

[1] As explained in section 2.4.1.1, *confidence score* is the numeric value created by the ASR, denoting how sure what was recognized is actually what the user said.

- **TASKCONSUME (ID=1)**: The agent tries to act on the task given by user (e.g. e.g. list all the clothing shops nearby). Does not cover the task to give directions.
- **GREET (ID=2)**: The agent greets the engaged user.
- **AGOODBYE (ID=3)**: The agent says goodbye to the user. Not to be confused with the *SGOODBYE* state attribute explained before (which determines whether the user is the one who said goodbye).
- **CHAT (ID=4)**: The agent goes into chat mode.
- **GIVEDIR (ID=5)**: The agent gives directions to a specific shop.
- **WAIT (ID=6)**: The agent waits for the next user utterance.
- **CONFIRM (ID=7)**: If the ASR confident score drops below 0.7, the agent asks for a confirmation of what the user said.
- **REQTASK (ID=8)**: The agent explicitly asks the user for a task.

The agent is generally unconstrained while taking actions (meaning most of the actions are available at any given moment for the agent to select). Though some actions are applicable only in certain states. Some state-action examples are show in table 5.1:

Table 5.1: Example of action restrictions

State attribute	**Available actions**
TURNTAKING = 1	WAIT

TURNTAKING = 0
TASKFILLED = 1
CTXTASK != null
	GREET, AGOODBYE, CHAT, REQTASK, TASKCONSUME

TURNTAKING = 0
TASKFILLED =1
CTXTASK = 'directions'
	GREET, AGOODBYE, CHAT, REQTASK, GIVEDIR

TURNTAKING = 0
LOWCONF = 1
	GREET, AGOODBYE, CHAT, CONFIRM

During the agent's turn (*TURNTAKING* state attribute is 0), the agent can take any action. But during the user's turn (*TURNTAKING* = 1) the agent can only take the *WAIT* action, effectively listening to the user. This explicitly captures the notion that the agent waits for a time frame after it has spoken, giving the time for the user to respond to an action.

5.1.3 Reward Function

At the end of each subsequent dialogue turn, the agent is rewarded (or punished) according to the action it chose to take in that specific turn (state). Since one of the objectives is to make the agent more engaging to the user, it is also rewarded for lengthier conversations and is calculated as follows:

+10 if the agent successfully completes a task (*TASKCOMPLETED* state attribute is set to 1 at the end of the turn).

+100 if the agent greets at an appropriate time (e.g. when it first starts engaging a user and not in the middle of the conversation).

+100 if the user terminates the conversation normally. Since it is practically impossible to imitate the exact reaction of a complacent user without some form of emotion detection (which is outside of this project's scope), the assumption was made that a content user would say goodbye before terminating the conversation with the agent (effectively setting *SGOODBYE* state attribute to 1).

-100 if the user abruptly leaves. This is the only punishment the agent gets and is applicable in numerous situations.

+1 for each subsequent turn, effectively accumulating to a big reward the more the user is engaged.

5.1.4 Termination Function

The termination function is quite simple. At the end of each turn, the agent checks whether the user is still engaged (*USRENGAGED* state attribute). If not, the conversation ends and the accumulated discounted reward is calculated.

5.1.5 Initial State

During training (and during live operation) it is paramount that the agent starts from a default state, called *initial state* in BURLAP. This state is:

- USRENGAGED = 1 (The interaction starts when the user enters approaches the agent)
- USRENGCHAT = 0
- LOWCONF = 0
- TASKCOMPLETED = 0
- USERTERMINATION = 0
- MODE = 0
- CTXTASK = *null*
- TIMEOUT = 0
- TASKFILLED = 0
- SGOODBYE = 0
- TURNTAKING = 0
- PREVACT = 0 (notice how there is no action with id = 0 in the action space. The value 0 implies it's the first turn and no previous actions have been taken).
- DISTANCE = A random integer number between 0 - 2 (This is due to the fact that we want the agent to learn how to react irrespective of the initial user's distance).

5.1.6 Learning Algorithm and Environment

In order for the agent to learn the optimal policy, a *learning algorithm* must be used, where the agent observes its *environment*, takes an action and it observes again the effects the action had on the environment. This is a challenging problem, since initially the agent has no knowledge of which action is better or about how the environment works. This is why a *simulated environment* was created in BURLAP, where the agent can run the learning algorithm over a large number of iterations, in order to converge to an optimal policy.

In this project, the *Q-Learning* algorithm was used to train the agent. For the Q-Learning algorithm to be used, some parameters must first be set. These are the *discount factor*, the *learning rate*, and the *initial q*. *Discount* was set to **0.99**, since the agent should care on delivering the correct responses (actions) in its turn immediately. For simplicity reasons, the *learning rate* was kept at a low fixed value (instead of implementing a decaying policy) at **0.1**. In order for the agent to explore as much as possible during the early stages of the training, the ϵ-*greedy* policy is followed with an *initial* ϵ of 0.9, decaying after each iteration i by a rate of $\frac{1}{i+1}$. The *q-values* were initialized at **0**. The agent was then left to train for 100.000 iterations. The training was also extremely fast, requiring less than one minute for the training to finish.

To evaluate the effectiveness of the training algorithm, an experiment was made where the agent performed the entire training over multiple trials. At the beginning of each trial, the agent was stripped of all previously gained knowledge, starting anew the training procedure.

Figures 5.1 and 5.2 show the average cumulative reward over 10 trials of 500 iterations and the average reward of the last trial of 500 iterations (episodes).

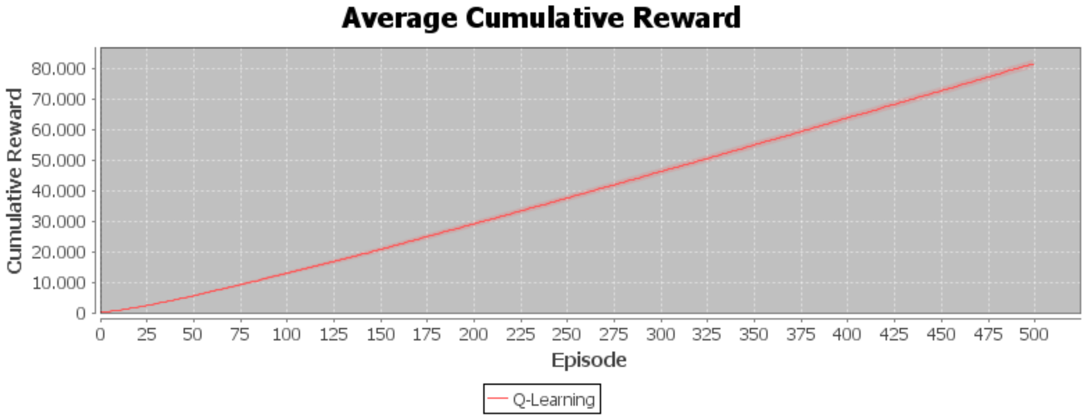


FIGURE 5.1: Average cumulative reward over 10 trials

Both these figures, illustrate the speed at which the algorithm trains the agent, where initially the agent gets a very low average reward, while accelerating rapidly.

After the training was complete, the policy was exported via a serialized Java object. This policy was then loaded and used throughout the evaluation process.

5.1.7 User Simulation

Since in the project's domain (user interaction within a shopping mall) the environment the agent must observe is the user, and no actual Human-Computer-Interaction (HCI)

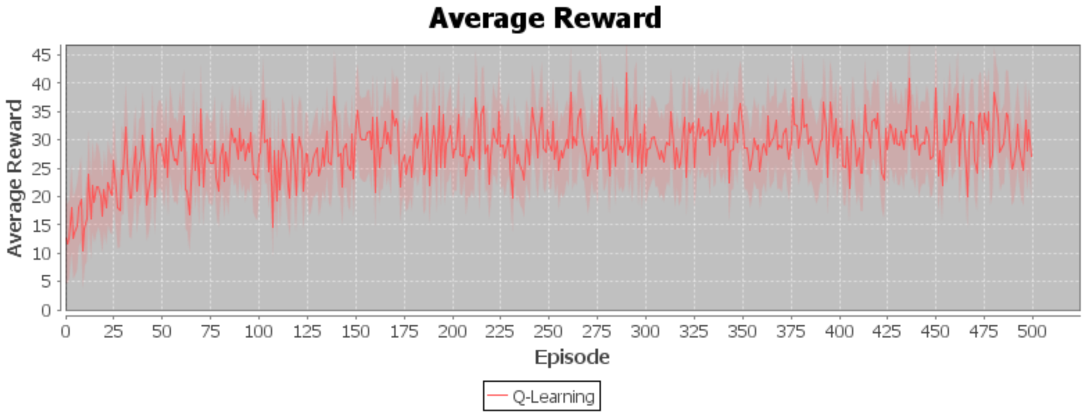


FIGURE 5.2: Average reward of a trial

data were collected from to formulate a predictive model, a simulated user had to be created, emulating how the user would most likely react to each action the agent takes [Rieser and Lemon, 2011]. It is clear by now, that the way q-learning works, the agent observes its current environment (reflected on the current state), takes an action (which leads into a new state) and then makes another observation of the new state. During this *transition* the user simulation operates. Since the agent and the user take actions (speak) in discreet turns, the user simulation "reacts" to the chosen action by the agent during his turn (TURNTAKING = 1). The user simulation effectively responds to the agent's action by changing the state attributes.

Since in this MSc project no actual user data was collected, in order to create a more realistic user behaviour, the user simulation operates based on assumptions on how an average user would act, set forth by the author.

The user simulation acts in the following way:

- If the agent greets in the middle of the conversation (taking the GREET action while it is not the first turn of the conversation), the user abruptly terminates the conversation by leaving (-100 reward).
- If the agent says goodbye before the user has, the user leaves abruptly (-100 reward).
- If the agent tries to perform a task given in the previous turn, but the confidence score is low (LOWCONF=1), assume that the task will not be completed correctly and the user leaves (-100 reward).

- If the user has given a task in his previous turn, and the agent does not try to perform that task, the user leaves (-100 reward).
- I the user starts leaving during his previous turn (see below) and the agent did not take an action to reengage him (Chat or ReqTask actions), then the user leaves (-100 reward)
- If the user has already said goodbye on HIS previous turn, he can leave normally (normal termination of episode/conversation).

If none of the aforementioned cases occur, the simulated user gets to a random distance (between 0.2 and 2.2 meters) from the agent and then bins it to close/medium/far (effectively updating the DISTANCE state attribute). After that he scholastically takes a random *user action* over a uniform probability distribution. These action are:

- **uGoodbye**: Where the user says goodbye to the agent [2]
- **uSilent**: The user stays silent
- **uWalkAway**: This action imitates a user starting to walk away from the agent (like when getting bored or annoyed), but is still within interaction range. This is done by selecting a random distance in the "far" distance (DISTANCE >1.8).
- **uChat**: The user chats with the agent (no task is given).
- **uReq_Task**: The user requests a specific task. The tasks are uniformly random between request for a coffee, an electronic device or clothing item.
- **uReq_Dir**: The user requests for directions.
- **uConfirm**: The user respond to a confirmation request by the agent.

Although these actions have normally equal probability of occurring during training, there are some factors that change this probability.

- If the **agent**'s last action was to request a task (because for instance the user did not speak during his last turn, or starts to walk away), the user has 80% probability of taking the *uReq_task* action, to assign a task to the agent.

[2] Remember that the assumption was made that if the user says goodbye before terminating the conversation, he is content.

- If the user is already engaged in chat with the agent, he has a 50% probability of continuing the chat, and 50% of taking another action.

In Table 5.2 a sample dialogue is given, in order to make the transition model clearer. In Table 5.3 the corresponding to that dialogue state transitions are illustrated.

TABLE 5.2: Example dialogue

(1) SYS: “Hi there”
(2) USR: “Hello. What is your name”
(3) SYS: “My name is Jarvis”
(4) USR: “My name is John”
(5) SYS: “Nice to meet you John”
(6) USR: “Do you know any nice place to get a cappuccino?”
(7) SYS: “Right. Give me a second. There are 2 coffee shops nearby. These are Starbucks, and Costa”
(8) USR: “Cool thanks! How can I get to Starbucks then?”
(9) SYS: “To get there you need to go left then straight ahead and you will find it after the second junction”
(10) *User starts moving away from the agent*
(11) SYS: “Is there anything I can help you with?”
(12) USR: “No thank you. Goodbye!”
(13) SYS: “Have a nice day!”
(14) *TERMINAL STATE*

Every other turn (whenever its the user’s turn) the agents takes the *Wait* action, where it just listens to the user. The user’s behaviour changes the environment (the state attributes), where it is observed at the end of that turn (e.g. when the user asks for a cappuccino during turn (6), the agent sees the updated state at the beginning of the next turn

5.2 IrisTK - Core Dialogue Manager

Both versions use IrisTK [Skantze and Al Moubayed, 2012] as core system, enriched with chatbot features. This is a Java based framework used to develop multimodal dialogue systems. IrisTK consist of interconnected modules, responsible for the various tasks in the life cycle of each turn. The interaction’s flow is defined using a statechart-based framework called *IrisFlow*. The flow uses the SCXML representation, based on the Harel model used by Al Moubayed et al. [2012], providing a more readable and well-structured code. This is then compiled into Java source code for execution.

TABLE 5.3: State transition in example dialogue

Distance	Mode	usrEngChat	timeout	lowConf	tskFilled	ctxTask	tskCompl	usrEng	sBye	usrTerm	turnTaking	actionTaken	Turn
1	0	0	0	0	0		0	1	0	0	0	Greet	(1)
1	0	0	0	0	0		0	1	0	0	1	Wait	(2)
1	1	1	0	0	0		0	1	0	0	0	Chat	(3)
1	1	1	0	0	0		0	1	0	0	1	Wait	(4)
1	1	1	0	0	0		0	1	0	0	0	Chat	(5)
1	1	1	0	0	0		0	1	0	0	1	Wait	(6)
1	1	0	0	0	1	coffee	0	1	0	0	0	taskConsume	(7)
1	0	0	0	0	0		1	1	0	0	1	Wait	(8)
1	0	0	0	0	1	directions	0	1	0	0	0	giveDirections	(9)
1	0	0	0	0	0		1	1	0	0	1	Wait	(10)
2	0	0	1	0	0		0	1	0	0	0	reqTask	(11)
2	0	0	0	0	0		0	1	0	0	1	Wait	(12)
2	0	0	0	0	0		0	1	1	0	0	goodbye	(13)
2	0	0	0	0	0		0	0	0	0	1	Wait	(14)

An abstract flow in XML looks like this [Iristk.net-overview, 2016]:

```
<flow name="FLOW_NAME" package="FLOW_PACKAGE" initial="INITIAL_STATE_ID">

    <!-- Zero or more flow-level parameters -->
    <param name="VAR_NAME" type="VAR_TYPE"/>

    <!-- Zero or more flow-level variables -->
    <var name="VAR_NAME" type="VAR_TYPE"/>

    <!-- One or more states -->
    <state id="STATE_ID">
        <!-- Zero or more state-level parameters -->
        <param name="PARAM_NAME" type="PARAM_TYPE"/>
        <!-- Zero or more state-level variables -->
        <var name="VAR_NAME" type="VAR_TYPE"/>
        <!-- Zero or one onentry event handler -->
        <onentry>
            <!-- actions to execute when the state is entered -->
        </onentry>
        <!-- Zero or more onevent event handler -->
        <onevent name="EVENT_NAME" cond="CONDITIONS">
            <!-- actions to execute when the event is received -->
        </onevent>
        <!-- Zero or one onexit event handler -->
        <onexit>
```

```
            <!-- actions to execute when the state is exited -->
        </onexit>
    </state>

</flow>
```

The platform is event-based, meaning that its whole functionality can be broken down to individual events that are generated and consumed. These events are mainly categorised into 3 groups, action, monitor and sense.

- Actions are events that enable the system to perform certain tasks, like speak, gesture, send another event, sleep, etc.
- Senses are events that represent what the system perceives, like when the user starts and/or stops speaking, when a user initiates or ends a conversation, when the user is silent, etc.
- Monitors are events that are tied to the action event that provides feedback concerning these actions. For instance when the system sends an event to start speaking (action.speech), the monitor.speech.end event will be raised when the utterance is done outputting, and the system should start listening to the user again via the action.listen event. [Skantze et al., 2014]

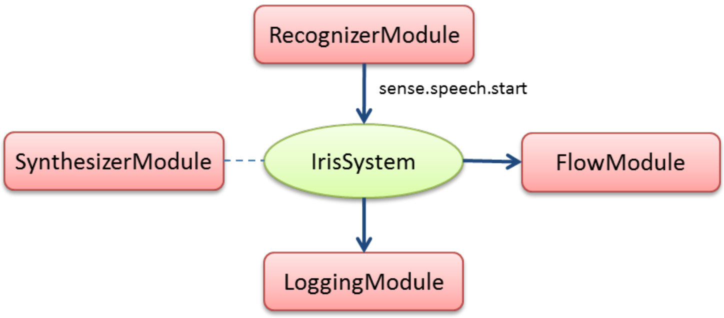


Figure 5.3: The IrisTK system and modules

In order for the IrisTK to know the environment it operates in, a Microsoft Kinect Sensor 2.0 was connected to the computer, where by using the skeletal tracking features and its depth camera, it is able to detect and track the interacted user. Although Kinect provides its own microphone array, for this project a Sony's SingStar microphone was used. IrisTK's module uses the Kinect sensor to identify and track the users, using the basic event *sense.body*, including head and hands detection, and head orientation, providing the location of each body part in 3D space.

For speech recognition (ASR), Nuance Cloud is used as an open vocabulary, since the system should be able to recognize (but not necessarily parse) almost anything the user is saying.

Along with the open vocabulary, a semantic grammar is used for this project in ABNF format. As the main focus of this project is not to design a complex dialogue system that is able to semantically parse a large amount of user utterances, the semantic grammar covers the understanding of a small amount of tasks. The parsing is constricted in the following categories:

- General greeting
- Request a specific item of one of the categories coffee, electronics or clothing.
- Request for directions, either using a generic phrasing like "I need some directions" or ask for directions to a specific shop
- Yes or No answers

Although the grammar is limited, effort has been made to be able to parse the above utterances using any possible phrasing, by detecting keywords inside the user utterance. For instance, in order to request a coffee, the user might say "Where can I buy a cup of coffee,", "I would like to buy a latte", "I want a cappuccino", etc., and all of those utterances will have the same effect on the system. All of the above mentioned product categories include several different objects (different types of coffee, different electronic devices or different clothing items). Also the system is able to parse a greeting or affirmative or negative phrase in different ways.

5.2.1 Handcrafted Rule Transition version

As explained in this thesis, two versions of the system were implemented. One using only handcrafted transition rules between states, where each subsequent agent's action is solely decided based on simple *if..else* rules, according to what was recognized by the

ASR in the Dialogue Manager, and another where the decisions were made by referring to a trained MDP policy.

In the baseline version of the system (handcrafted rules), the dialogue manager uses slotfilling technique to perform the tasks at hand.

Each of the grammar parsed expressions are automatically bound to JavaScript objects (also called tags) in order to pass information between the different system components. For instance, when the user uses any of the recognised words for coffee (cappuccino, latte, mocha, etc.), the system sets the tag productType as "coffee". When the user asks for directions (again using one of the available expressions), the flag tag *req_direction* is set to true and the tag *dest* is either filled with the shop name the user have spoken, or is left empty (in which case it will be asked on a later turn by the system). These JavaScript objects will later be adjoined as IrisTK Record objects in the Dialogue Manager.

The decisions on each turn are taken by the IrisFlow module, where the different possible states[3] are defined. There, three generic states reside. One when the system is not engaging a user, called *Idle*, the second that handles the user interaction, called *Dialog* and the third, that the system returns after each turn waiting for the next user input, called *Await*. There are also states that handle tasks, give directions, initiate chat, etc. From the Dialog state, the system branches out to all the rest sub-states, by having the *Dialog* state pick up the user phrase, and according to the semantic parsing, the flow goes to the corresponding state where the agent's response is synthesized. For instance, when the user request directions, the *req_directions* state is called, and acts appropriately depending on whether the user has already provided the place where he needs directions to, or whether this information is omitted from the user utterance and must be asked by the system. These transitions are possible via the *sense.user.speak* event, along with the semantic parsing of the user utterance.

The baseline system uses a basic rule to distinguish between a task request and chatting. If the user says something that can be parsed through the semantic grammar (using phrase and keyword detection), then the corresponding IrisTK state is invoked to perform the task (as explained below). If not, then the user utterance (bound to

[3] The IrisTK states are not to be confused with the MDP states. These represent the state where the flow currently is, while the MDP states represent what the environment looks like at any given time.

the IrisTK *event*) is passed to the *SentToBot* state, effectively letting the Chatbot try respond to the user (see section 5.3).

For task handling, a slot filling parsing technique is used (see section 2.4.1.4), where the dialogue system needs to get a specific set of information from the user before completing a task. In the "request direction" example above, the information needed to complete the task (called slot) is the name of the shop. The designed system uses slots in two occasions: either to provide directions (where the shop's name is the slot), or to provide the shops' names that sell a specific item (where the item's name is the slot), or using a combination of both. This is achieved by binding specific words said by the user to two IrisTK Record object variables (productType and req_directions), as described previously. The relevant information from the grammar (the *tags*) and the name of the shop, are ad-joined in the Dialogue Manager into the *req_directions* object, and passed to the Java class *ShopList*.

For the needs of this project, as the amount of data required is not large, a simple text file *shop_list.txt* is used to act as a database. This file contains a list of all available shops in comma separated format. Each shop contains the following information:

- shopName: The name of each shop
- category: 3 categories are supported. Clothing, electronics and coffee shops.
- directions: A preset directions string relative to the agent's position in the shopping mall. This also includes the delay required for lip synchronisation.

This ShopList class operates as the backend for several operations. At system startup, it reads the *shop_list.txt* file, and populates an ArrayList collection, used in all other occasions. For instance, if the user requests directions, the operation proceeds as follows:

After the req_directions Record object has been created (by adjoining the information from the grammar), the CheckDestination state is called, where a check is made to see whether all required information to complete the tasks have been filled. If not, the agent prompts the user for the missing information. Then the name of the shop passes to the ShopList class, where a filtered collection of shops is enumerated, holding only the shops that fall under the requested category. The size and contents of this filtered list is first

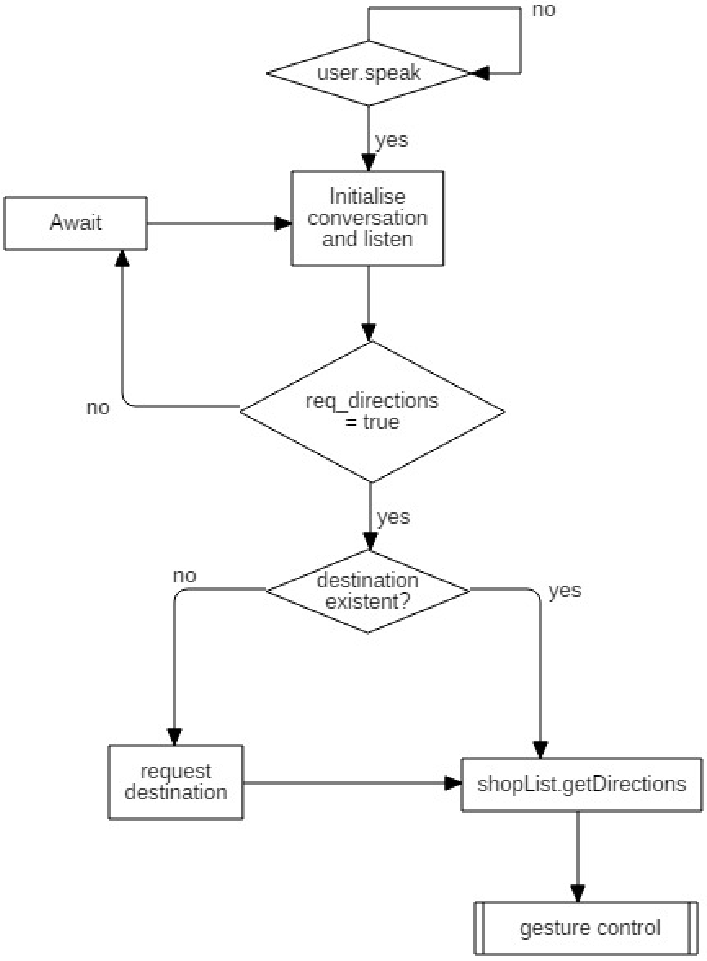


Figure 5.4: Simple directions request flowchart

then returned to the user, constructing the system's utterance: "*There are <size of filtered list><category of shop>shops nearby. These are <enumeration of contents>.*"

The code below illustrates how the main state *Dialog* transitions to the *consumeQuestion.* If the user speaks, and he request for e.g. a coffee, a Record object is created by adjoining the semantic information extracted from the grammar, and the flow goes to the consumeQuestion state, after which the agent responds accordingly. Then the flow returns to the *Await* state.

```
<state id="Dialog">
    <onevent name="sense.user.speak" cond="event?:sem:productType">
        <exec>productType.adjoin(asRecord(event:sem:productType))</exec>
        <goto state="consumeQuestion" />
    </onevent>
```

```
</state>


<state id="consumeQuestion" extends="Dialog">
        <onentry>
                <exec>shop=shopList.getShop(req_directions:dest)</exec>
                        <random>
                                <agent:say>Let me see.</agent:say>
                                <agent:say>Well,</agent:say>
                                <agent:say>I think I know what you are
                        looking for.</agent:say>
                                <agent:say>Let me search that for you.</agent:say>
                                <agent:say>Right. Give me a second </agent:say>
                        </random>
                        <agent:say>There are <expr>shopList.
                filteredCategory(asString(productType:type)).size()</expr>
                <expr>asString(productType:type)</expr> shops nearby</agent:say>
                        <agent:say>These are<expr>shopList
                .filteredCategory(asString(productType:type)).
                enumShops()</expr></agent:say>
                        <goto state="Await"/>
        </onentry>
</state>
```

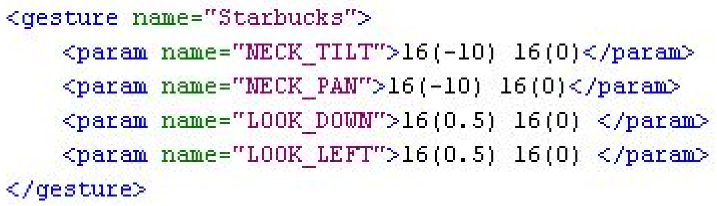
Listing 5.1: Dialog and consumeQuestion states (Handcrafted version)

In case the system is unsure on what exactly the user has said (when the confidence value drops below 0.7), the system calls the Confirmation state, in order to confirm the last user utterance. In order for the confirmation dialogue to not repeat the complete last user phrase, each slot value is bound to a secondary tag (out.confirm = meta.*.text) in order for the confirmation to be for example *"Did you say Coffee Time"* instead of *"Did you say how can I go to Coffee Time"*.

5.2.1.1 Simulation and gesture control

When the user requests directions, and all required information slots have been filled, the Dialogue Manager retrieves the direction String for the requested shop from the database, by invoking the *getDirections()* method of the *ShopList* class. This String representation of the directions, consist of both the actual (predefined) directions as well as the appropriate delay for lip syncing. This information is bound to a global variable used by the Dialogue Manager. Then the *fixateAt* state is called, where the directions string is separated into its components (actual directions part and delay part). After this two separate action events are sent to the IrisSystem, the *action.speech* that calls the Synthesizer to output the direction string, and the *action.gesture* to perform a head gesture according to the directions.

The simulated human head performs a left or right nodding, according to the directions it gives (meaning each time the agents says the word "left" or "right", the head is nodding in this direction, while keeping its gaze on the user, simulating the human's nodding. Each shop is tied to a unique gesture movement. These gestures are defined using the IrisTK Gesture Builder, an interface of the IrisTK platform. These are coded using XML.

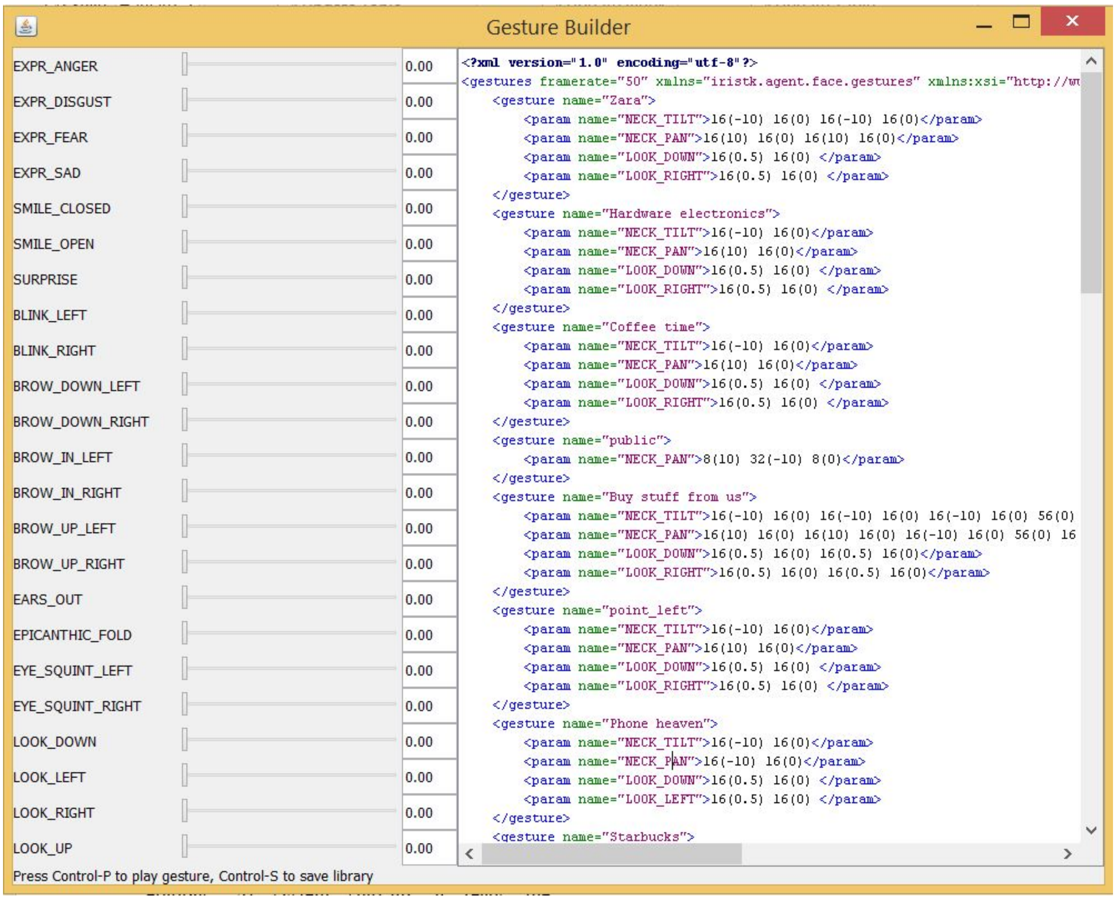


```
<gesture name="Starbucks">
    <param name="NECK_TILT">16(-10) 16(0)</param>
    <param name="NECK_PAN">16(-10) 16(0)</param>
    <param name="LOOK_DOWN">16(0.5) 16(0) </param>
    <param name="LOOK_LEFT">16(0.5) 16(0) </param>
</gesture>
```

FIGURE 5.5: Gesture format in XML

Each gesture has a specific name and is broken down to individual small gestures and trajectories. In this project, each shop has its gesture named after the shop's name, in order to make the retrieval of gestures easier. In the example shown in Figure 5.5, by executing this gesture, the animated head will nod to its right, by tilting its head, panning the neck and looking down and left. Each of these sub-gestures have trajectories, specifying the start and end position, in relevance to the previous position, and the speed required to get into that position (noted in frames, where each frame is 1/50 of a second) [Iristk.net-gestures, 2016]. In the given example, the neck will pan from the initial position 0, to position 10 in 16 frames, and then return to position 0 in another 16 frames.

In order to synchronize the head movement with the vocalisation of the words "left" and "right", a delay is required between each event (speech and gesture). This delay is also hardcoded in the database, tied to each direction, and passed along as a parameter in the send event="action.gesture" command.



EXPR_ANGER	0.00
EXPR_DISGUST	0.00
EXPR_FEAR	0.00
EXPR_SAD	0.00
SMILE_CLOSED	0.00
SMILE_OPEN	0.00
SURPRISE	0.00
BLINK_LEFT	0.00
BLINK_RIGHT	0.00
BROW_DOWN_LEFT	0.00
BROW_DOWN_RIGHT	0.00
BROW_IN_LEFT	0.00
BROW_IN_RIGHT	0.00
BROW_UP_LEFT	0.00
BROW_UP_RIGHT	0.00
EARS_OUT	0.00
EPICANTHIC_FOLD	0.00
EYE_SQUINT_LEFT	0.00
EYE_SQUINT_RIGHT	0.00
LOOK_DOWN	0.00
LOOK_LEFT	0.00
LOOK_RIGHT	0.00
LOOK_UP	0.00

```
<?xml version="1.0" encoding="utf-8"?>
<gestures framerate="50" xmlns="iristk.agent.face.gestures" xmlns:xsi="http://ww
    <gesture name="Zara">
        <param name="NECK_TILT">16(-10) 16(0) 16(-10) 16(0)</param>
        <param name="NECK_PAN">16(10) 16(0) 16(10) 16(0)</param>
        <param name="LOOK_DOWN">16(0.5) 16(0) </param>
        <param name="LOOK_RIGHT">16(0.5) 16(0) </param>
    </gesture>
    <gesture name="Hardware electronics">
        <param name="NECK_TILT">16(-10) 16(0)</param>
        <param name="NECK_PAN">16(10) 16(0)</param>
        <param name="LOOK_DOWN">16(0.5) 16(0) </param>
        <param name="LOOK_RIGHT">16(0.5) 16(0) </param>
    </gesture>
    <gesture name="Coffee time">
        <param name="NECK_TILT">16(-10) 16(0)</param>
        <param name="NECK_PAN">16(10) 16(0)</param>
        <param name="LOOK_DOWN">16(0.5) 16(0) </param>
        <param name="LOOK_RIGHT">16(0.5) 16(0) </param>
    </gesture>
    <gesture name="public">
        <param name="NECK_PAN">8(10) 32(-10) 8(0)</param>
    </gesture>
    <gesture name="Buy stuff from us">
        <param name="NECK_TILT">16(-10) 16(0) 16(-10) 16(0) 16(-10) 16(0) 56(0)
        <param name="NECK_PAN">16(10) 16(0) 16(10) 16(0) 16(-10) 16(0) 56(0) 16
        <param name="LOOK_DOWN">16(0.5) 16(0) 16(0.5) 16(0)</param>
        <param name="LOOK_RIGHT">16(0.5) 16(0) 16(0.5) 16(0)</param>
    </gesture>
    <gesture name="point_left">
        <param name="NECK_TILT">16(-10) 16(0)</param>
        <param name="NECK_PAN">16(10) 16(0)</param>
        <param name="LOOK_DOWN">16(0.5) 16(0) </param>
        <param name="LOOK_RIGHT">16(0.5) 16(0) </param>
    </gesture>
    <gesture name="Phone heaven">
        <param name="NECK_TILT">16(-10) 16(0)</param>
        <param name="NECK_PAN">16(-10) 16(0)</param>
        <param name="LOOK_DOWN">16(0.5) 16(0) </param>
        <param name="LOOK_LEFT">16(0.5) 16(0) </param>
    </gesture>
    <gesture name="Starbucks">
```

Press Control-P to play gesture, Control-S to save library

FIGURE 5.6: The IrisTK gesture builder

5.2.2 MDP policy version

While the decisions were taken using a single simple rule in the Handcrafted policy version, presented in Chapter 5.2.1, and letting the Dialogue Manager transition from state to state, this version tries to substitute this single rule, with a more complex model for deciding the next action, by using a trained MDP policy.

To achieve this, an interface between IrisTK and BURLAP was created using Java (*MDPTrainer.java*), where it initially recreates the *domain* in which the MDP was trained, as shown in section 5.1, loads the serialized *policy* from file. In this version, all action control from*IrisFlow* has been transferred to the *MDPTrainer*. This means that although the states in IrisFlow are the same (Dialog, Await, Idle, SendToBot, etc.), all hardcoded transition between them has been removed (So the *Dialog* state no longer calls the corresponding states according to the semantic representation of the user utterance, as was occurring in the previous version). Instead, an additional core state has been added to the flow, called *ObserveState*, where the flow goes after each action (turn). In order for the *IrisFlow* and *MDPTrainer* to share information, all state attributes (as shown in 5.1.1) are now also present in IrisFlow as internal variables. The updated IrisFlow works like this:

On each turn, the *ObserveState* state is called, where it passes the MDP state attributes to the *MDPTrainer*. There, a BURLAP State object is created using those values, and after a lookup in the loaded policy, the action id with the highest Q-Value for that MDP state is returned to the IrisFlow. According to that id, the *ObserveState* then calls the corresponding IrisTK state to generate and output the agent's response. While previously after each turn, the flow returned to the *Await* IrisTK state, now it always returns to the *ObserveState*, since the *MDPTrainer* is responsible for deciding whether the agent should "wait" for the user to speak or not.

The example code shown in the Handcrafted version, now works like this when adapted to use the MDP:

```
<state id="Dialog">
    <onevent name="sense.user.speak" cond="event?:sem:productType">
        <exec>productType.adjoin(asRecord(event:sem:productType))</exec>
        <exec>taskFilled = true; ctxTask = asString(productType:type)</exec>
    </onevent>
</state>


<state id="consumeQuestion" extends="Dialog">
        <onentry>
                <exec>shop=shopList.getShop(req_directions:dest)</exec>
                        <random>
                                <agent:say>Let me see.</agent:say>
                                <agent:say>Well,</agent:say>
                                <agent:say>I think I know what you are
                        looking for.</agent:say>
                                <agent:say>Let me search that for you.</agent:say>
                                <agent:say>Right. Give me a second </agent:say>
                        </random>
                        <agent:say>There are <expr>shopList.
                filteredCategory(asString(productType:type)).size()</expr>
                <expr>asString(productType:type)</expr> shops nearby</agent:say>
                        <agent:say>These are<expr>shopList
                .filteredCategory(asString(productType:type)).
                enumShops()</expr></agent:say>
                        <return/>
        </onentry>
</state>
```

Listing 5.2: Dialog and consumeQuestion states (MDP version)

Notice in the *Dialog* state, the flow does not directly go to the *consumeQuestion* state, rather it just sets the *MDP state attributes* accordingly. When the flow continues its normal execution (since no *goto state* command redirects the flow to a certain state), the *MDPTrainer* dictates *ObserveState* to call indirectly the *consumeQuestion* state.

5.3 Program AB - Chatbot

Program AB [ALICE AI Foundation, 2013] is a JAVA implementation for the AIML 2.0 specification. It provides an easy way to implement, compile and run chatbots based on the A.L.I.C.E. bot. A bot created with Program AB, follows the following file structure in order to be compiled and run correctly:

- /bots/botname/aiml : This folder includes the AIML files
- /bots/botname/aimlif : This folder includes the AIMLIF files
- /bots/botname/config : This folder includes the bot configuration files
- /bots/botname/sets : This folder includes the set files
- /bots/botname/maps : This folder includes the map files. Map files are used to map certain words to more generic ones (e.g. the word “sis” is mapped to the word “sister”)

Program AB comes along a predefined sample bot called *S.U.P.E.R.* [ALICE AI Foundation, n.d.], which was used in this project as a baseline, altered in order to be appropriate for the shopping mall domain. Normally bots created with this software can be interacted with only via text input/output. Since *IrisTK* can easily export any user utterance (since the user utterance is bound to the *event* object) to text, communication between the chatbot and IrisTK was very easy.

A simple JAVA class was devised for the purposes of this project to act as an interface between the chatbot and the IrisTK. *Program AB* provides an easy way to interact with the bot via Java code, by creating a *Chat* object which will hold the chatting session. This allows the Chat object's *multisentenceRespond(String question)* to be invoked, which return the response of the chatbot. This allows IrisTK to pass any user utterance that is not parsed by the semantic grammar through to this method and return the bot's response as a *String*, allowing easily IrisTK TTS module to synthesize the response to voice.

Chapter 6

Evaluation and Results

6.1 Experimental design and Evaluation

As explained before, to evaluate the usefulness and performance of the proposed system, it was evaluated against a baseline system, using handcrafted rules for action selection.

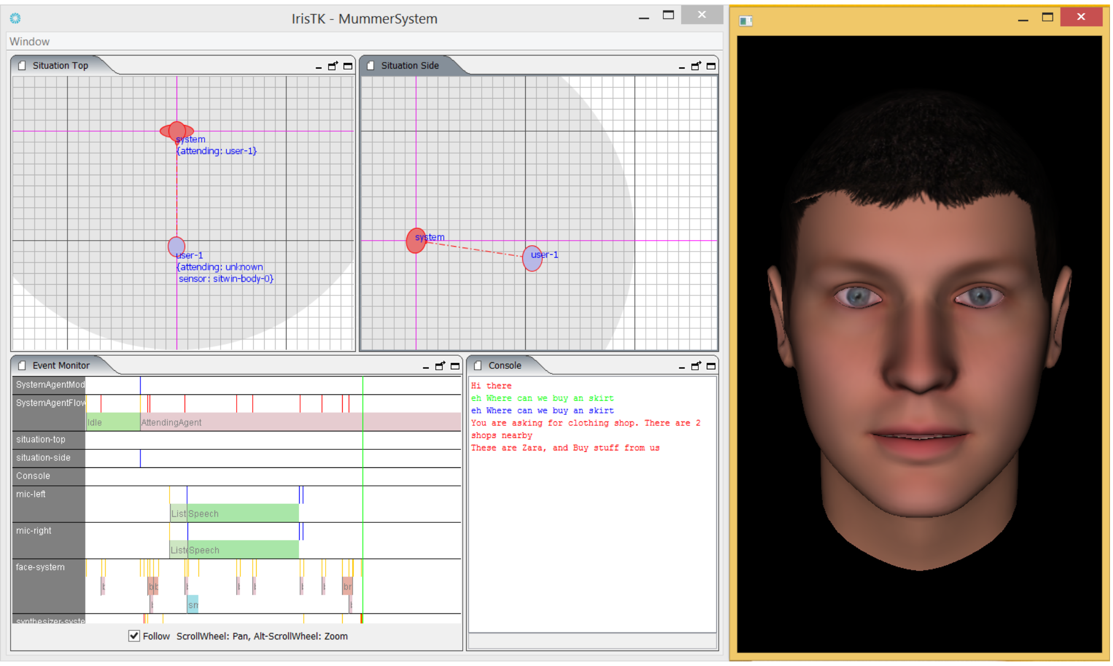
The independent variables (IV) in this evaluation are the two system variants (with and without the MDP module). The dependent variables (DV) include:

- Usability
- user engagement
- User preference
- Natural dialogue flow
- Task success rate
- Entertaining factor
- Overall usefulness of the system

To test these variables, a test population of 10 subjects was used, consisting of 4 females and 6 males, of mixed nationalities, ranging from 20 to 30 years of age. They have been introduced to both versions of the system (in random order and without them having any prior knowledge on each system's capabilities) following the *within-subject* experimental design, by interacting with the IrisTK *simulated avatar*, after which they were asked to fill in a questionnaire, as shown in Appendix C. The questions are using a 6 point *Likert scale* and are as follows:

- Q1. I found the agent engaging
- Q2. The agent behaved as expected
- Q3. The dialog flow was natural
- Q4. The agent understood my commands well
- Q5. The agent provided sensible answers
- Q6. The conversation was enjoyable / fun
- Q7. I found the agent uncanny / creepy
- Q8. The agent managed to keep my interest in chatting with it, even when it misbehaved

with 1 meaning that they agree completely (most positive) and 6 meaning they disagree completely (most negative).



```
Hi there
eh Where can we buy an skirt
eh Where can we buy an skirt
You are asking for clothing shop. There are 2
shops nearby
These are Zara, and Buy stuff from us
```

FIGURE 6.1: The IrisTK simulation environment

Apart from the user survey, the system automatically logged the length of each participant's conversation in seconds, the dialogue itself (so the number of turns can be extracted), as well as the transition states in the MDP version of the system.

In order for the ASR to perform adequately, a *Sony SingStar* Wireless microphone was used. The participant had to enter the Kinect's field of view to initiate the experiment, and had been given enough space to be able to change his/her distance from the sensor at any given time during the conversation.

The participants were asked to request the agent to perform 3 tasks: To ask for a *coffee*, ask for an *electronic* device and ask for *directions* to one of the shops the agents responds with. The participants were not instructed to use any specific phrasing or keywords, rather to formulate the request in their own words, in order to test the agent's understanding capabilities within a shopping mall. They were also not instructed explicitly to chat with the agent, since it was the author's hope that these features will emerge naturally within the conversation flow.

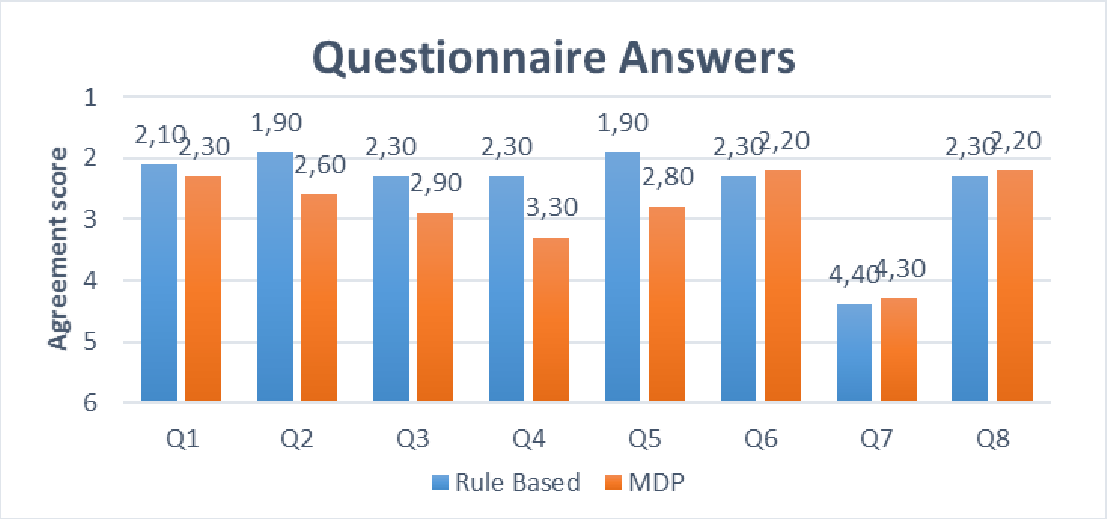


Figure 6.2: Questionnaire answers average score (higher is better: 1 = agree, 6 = disagree)

From a quick review of the questionnaire answers, as shown in Figure 6.2, both versions performed almost equally, with the Rule-based version (handcrafted policy) scoring slightly higher in almost all areas. Given the subjective nature of the questions, as well as the amount of participants in the evaluation, some results are not conclusive. For instance, Q4 that refers to the command understanding, shows the biggest difference among the questions. Since both versions of the system were using the same

SLU though, this question was expected to show almost identical results. Yet, a trend is observed where Q5 directly affects the results of Q4. Despite this fact, none of the questions above were deemed significant, as shown in the *significance results* in Tables 6.1 and 6.2 (T-tests), as well as in Tables 6.3a and 6.3b (Chi-Square tests).

It is also worth noting that the questions where the MDP version surpassed the rule-based one are Q6, Q7 and Q8, all of them tied with the anthropomorphic and engaging attributes of the agent. This is best reflected in Figures 6.3 and 6.4, illustrating the average number of turns of each conversation and the time spend on each conversation respectively.

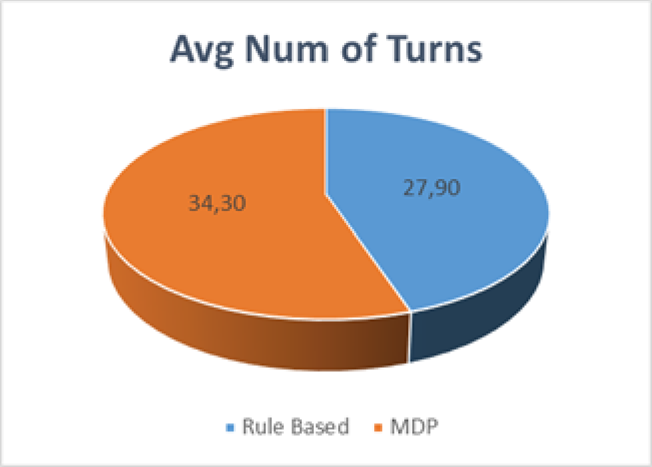


FIGURE 6.3: Average turns per dialogue

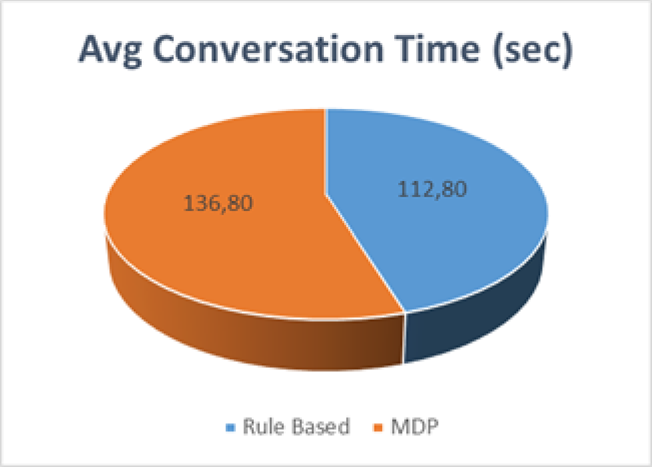


FIGURE 6.4: Average time per dialogue in seconds

It is clear from these graphs that the MDP version outperformed the rule-based version in terms of more engaging and lengthier conversations, which is to be expected, since this is one of the aspects the MDP *reward function* is optimizing for. On the other hand, task completion was higher in the rule-based version, result that is consistent with the overall outcome shown in Figure 6.2.

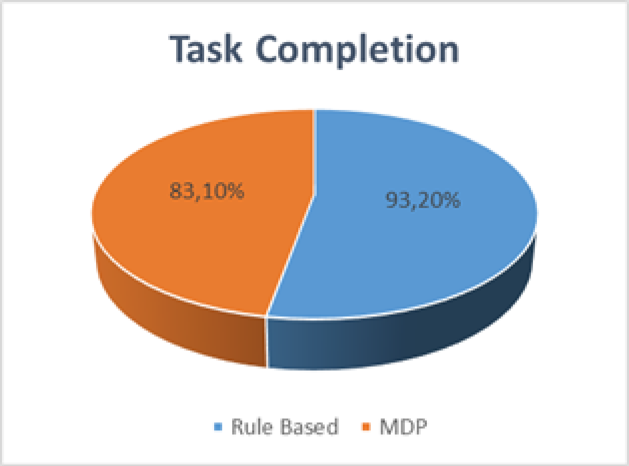


FIGURE 6.5: Task completion

TABLE 6.1: t-test for average time per dialogue

T-test (Avg Time)		
Alpha	0,05	
Hypothesized Mean Difference	0	
	MDP	**Rule-Based**
Mean	136,80	112,80
Variance	5437,51	3138,18
Observations	10,00	10,00
Pearson Correlation	0,59	
Observed Mean Difference	24,00	
Variance of the Differences	3732,22	
df	9,00	
t Stat	1,24	
P (T¡=t) one-tail	0,12	
t Critical one-tail	1,83	
P (T<=t) two-tail	0,25	
t Critical two-tail	2,26	

Table 6.2: t-test for average turns per dialogue

T-test (Avg Turns)		
Alpha	0,05	
Hypothesized Mean Difference	0	
	MDP	**Rule-Based**
Mean	34,30	27,90
Variance	402,68	200,99
Observations	10,00	10,00
Pearson Correlation	0,62	
Observed Mean Difference	6,40	
Variance of the Differences	248,93	
df	9,00	
t Stat	1,28	
P (T¡=t) one-tail	0,12	
t Critical one-tail	1,83	
P (T¡=t) two-tail	0,23	
t Critical two-tail	2,26	

Table 6.3: Independence tests (Chi-Square)

(a)

Test of Independence (Chi-Square) for Q4	
Alpha	0,05
df	9
P-value	0.97
Test Statistic	2.68
Critical Value	16,92

(b)

Test of Independence (Chi-Square) for Q5	
Alpha	0,05
df	9
P-value	0.95
Test Statistic	3.27
Critical Value	16,92

6.2 Example Dialogues

Below an example dialogue is shown, as produced during evaluation:

system : Hi there

user : Hello

system : Hi how are you?

user : I'm fine thank you do you know where I can drink coffee

system : Right. Give me a second. There are 2 coffee shops nearby. These are Starbucks, and Costa.

USER : How can I go to the Starbucks?

SYSTEM : To get there you need to go left then straight ahead and you will find it after the second junction

USER : OK thank you

SYSTEM : You're welcome

USER : Do you know where I can find a charger for my phone

SYSTEM : I think I know what you are looking for. There are 3 electronics shops nearby. These are Hardware electronics, Public, and Game

USER : Which one is better

SYSTEM : The first one is usually the best one

USER : (inaudible)

SYSTEM : Are you there?

USER : Okay thank you

SYSTEM : It was my pleasure.

Here a basic conversation was carried out, executing the 3 tasks as described in section 6.1. During most of the turns, the agent was in *task mode*, responding to the tasks given one after the other. It is noticeable that when the user asked *"Which one is better"*, which is an utterance with no semantic representation according to the semantic grammar (see Appendix A), the agent fired the chatbot to produce the response *"The first one is usually the best one"*. Also near the end of the conversation, the user said something (so no *silence* event was raised) but the ASR was not able to recognize at all, so the agent fired the chatbot again to respond with one generic engaging question: *"Are you there?"*.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this MSc project a hybrid Conversational Agent was implemented, combining a Dialogue Manager with a Chatbot. The system also uses multimodal sensors (distance, using a camera system), provides multimodal output with an animated face, and uses an action-selection policy trained using Reinforcement Learning (RL). This is the first dialogue system to combine chat and task-based dialogue using an RL controller. The system is acting based on a trained MDP policy, and evaluated against a version of the same system using a handcrafted policy.

The results discussed in chapter 6 provide evidence that this system performs better in some respects when using rule-based state transitions, rather than using reinforcement learning, although the version of the system using the MDP policy out-shined its combatant in terms of conversation duration. This outcome is to be expected though, given the simplicity of the problem the MDP had to solve, since the action space consisted of only 8 actions, where the state space produces a rather small size of state-action pairs (∼780.000).

The project also succeeded in covering all the mandatory requirements shown in Chapter 3, and some of the optional ones as well.

Specifically, it succeeded in:

1. **Being entertaining:** via the chatbot features incorporation to the SDS.

2. **Being generic within the domain:** The system is able to understand and extract the semantic representation of numerous utterances, irrespectively of how they were said, by extracting keywords with specific semantics.

3. **Having a natural dialogue flow:** Although the task-based responses were somewhat predefined, evaluation showed that interacting with the agent was feeling natural.

4. **Supporting mixed initiative:** The agent is able to either let the user initiate the conversation, or take the initiative and start the conversation itself.

5. **Able to switch between modes:** Via Reinforcement Learning training, the agent was capable on discerning whether it should perform a specific task (task-based mode), or chat with the engaged user (chat mode).

6. **Is trainable:** The agent is trained using Reinforcement Learning (a MDP model) to decide which action to take on each turn.

It also succeeded in some optional objectives as well:

1. **Being humorous:** The agent is able to tell jokes on demand via the chatbot.

2. **Response time:** The respond time was kept low, though no barge-in functionality was implemented.

In conclusion, it is the author's belief that chatbot-dialogue manager hybrid systems can be extremely engaging dialogue systems. Although a simple hybrid system can be implemented using simple transition rules, more complex systems that deal with a level of uncertainty or systems with heavy multimodal input (as discussed in section 7.2) should be trained using a MDP or POMDP.

7.2 Future Work

Although this project provides enough evidence on basic implementation and training of a hybrid chatbot-dialogue manager system, more research could be done by making the problem more complex. Just by adding the raw distance value to the state representation (without first binning it) would make the state space *continuous*, with almost infinite state-space. Then a function approximation method like *linear function approximation* [Bhatnagar, 2015, Xu et al., 2014] would be used to solve the MDP.

Also more sensory input could be provided, like basic emotion classification, engaged user age classification (being able to discern whether the engaged user is a child or an adult, and respond accordingly), head pose estimation and gaze detection, while simultaneously substituting the MDP model with an *POMDP*, to accommodate for the uncertainty in the environment. The EMOTE [EMOTE-Project, n.d.] project modules could be used for such a task. The increasing complexity of such a system would make also hand-crafting of the interaction rules infeasible, so that an RL method as presented here would be an attractive solution.

The agent could also be embodied into an actual robot, like the FurHat robotic head [Al Moubayed et al., 2012] or PEPPER Robot [Byford, 2014].

Appendix A

Semantic Grammar

#ABNF 1.0 UTF-8;

language en-US; root $root;

public $root = (([$greeting] [$filler] $productType out.productType = rules.productType; out.confirm = meta.productType.text [please]) — ($yes out.yes=1) — ($no out.no=1) — ($req_info out.req_info=1) — ($req_directions out.req_directions=rules.req_directions) — ($goodbye out.goodbye=rules.goodbye) — ($req_closing out.req_closing=rules.req_closing));

$productType = (($coffee out.type=”coffee”) — ($electronics out.type=”electronics”) — ($clothing out.type=”clothing”));

$buy = ((buy) — (get) — (find) — (have) —(drink));

public $req_directions = ([$greeting] (I need [some] directions out.dest=”null”) — (How can I ((get) — (go)) to $shops out.dest=rules.shops; out.confirm=meta.shops.text) — (Where can I find $shops out.dest=rules.shops; out.confirm=meta.shops.text) — ([Can you] Show me the way to $shops out.dest=rules.shops; out.confirm=meta.shops.text) — (Guide me to $shops out.dest=rules.shops; out.confirm=meta.shops.text) — (Where is $shops out.dest=rules.shops; out.confirm=meta.shops.text) — (direct me to $shops out.dest=rules.shops; out.confirm=meta.shops.text) — ([$shops out.dest=rules.shops; out.confirm=meta.shops.text]));

public $req_info = ([$greeting] (I need [some] information) — ($det have some questions) — (I want to ask you something) — (Can I ask you ((a [few] question?) — (something))));

public $coffee = [drink] [cup of] ((coffee) — (cappuccino) — (latte) — (mocha) — (espresso));

$electronics = (([mobile] phone) — (charger) — (TV) — (television) — (iphone) — (LG) — (Samsung) — (USB stick) — (phone covers) — (laptop) — (cable [for my phone]) — ([personal] computer));

$clothing = ((shoes) — (jacket) — (t-shirt) — (skirt) — (belt) — (jeans) — (trousers) — (socks) — (shirt) — (suit) — (coat) — (underwear) — (lingerie) — (clothing [shop]));

$det = ((I) — (we));

$article = ((a) — (an) — (some));

$greeting = ((Hello) — (Hi) — (Hi there) — (Good $partOfDay)) out.req_greeting=1;

$partOfDay = ((morning) — (afternoon) — (day) — (night));

$number = ((one out=1) — (two out=2) — (three out=3) — (four out=4) — (five out=5) — (six out=6) — (seven out=7) — (eight out=8) — (nine out=9) — (ten out=10));

$yes = ((yes) — (yes I do) — (sure) — (yeah) — (of course) — (okay));

$no = ((no) — (no way) — (nope) — (not really) — (I don't think so));

$filler = ((ahm) — (ah) — (eh) — (ehm) — (mm) — (can you (tell me) — (show me)));

public $shops = ((Starbucks) — (Hardware electronics) — (Costa) — (Tesco) — (Public) — (Primark) — (Phone heaven));

public $goodbye = ([OK] ((Bye) — (Goodbye) — (See you [later]) — (Cheers)));

Appendix B

shop_list.txt

shopName;category;subcategory;directions;sales

Starbucks;coffee;;To get there you need to go left then straight ahead and you will find it after the second junction - 1000;false

Hardware electronics;electronics;;To get there you need to go to my right then straight ahead past Coffee Time after the second junction - 1500;false

Costa;coffee;;To get there you need to go to the right corridor and its the third shop you will meet. - 1500;false

Tesco;clothing;;To get there you need to go directly to my right then turn right on the first junction - 1800;true

Public;electronics;;To get there you need to go on either corridor and it is the last shop you will find - 1200;false

Primark;clothing;;To get there you need to go directly to my right then turn right and then left and it is the first shop on your left - 1800;false

Phone heaven;electronics;;To get there you need to go left then its the third shop - 1500;true

Appendix C

Evaluation Form

Human-Robot Interaction
(System Evaluation Form)

Research performed by: Ioannis Papaioannou, Artificial Intelligence MSc – Heriot Watt University, Edinburgh

During this experiment, you will be asked to talk to the digital avatar/robotic head following these instructions:

- Please imagine that you are inside a shopping mall that you have never been to before, where the agent will be installed in an entry point, engaging visitors one at a time.
- To start the evaluation, enter the field of view of the camera. You can see your face and hands detected and tracked on the computer screen in front of you.
- The agent is able to chat and to help you find places in the mall (e.g. a coffee shop)

Please <u>do not simply read out the tasks below</u> to the agent – but try to express them in your own words. Try to complete the following tasks:

<u>TASK 1</u>

You want to find a place where you can drink a coffee Completed Y/N

<u>TASK 2</u>

You want to find a place to buy an accessory for your phone (e.g. a charger).
Completed Y/N

<u>TASK 3</u>

Get directions to one of the aforementioned shops. Completed Y/N

- Now please answer the questions overleaf -

Your information and responses will remain anonymous and confidential. The data collected will be only be reported as a collective combined total and only the researcher will have knowledge of your individual answers.

QUESTIONNAIRE

Age:
Gender:

For each item identified below, circle the number
to the right that best fits your judgment of its quality.
Use the rating scale to select the quality number.

Survey Item		Scale				
	Agree					Disagree
1. I found the agent engaging	1	2	3	4	5	6
2. The agent behaved as expected	1	2	3	4	5	6
3. The dialog flow was natural	1	2	3	4	5	6
4. The agent understood my commands well	1	2	3	4	5	6
5. The agent provided sensible answers	1	2	3	4	5	6
6. The conversation was enjoyable / fun	1	2	3	4	5	6
7. I found the agent uncanny / creepy	1	2	3	4	5	6
8. The agent managed to keep my interest in chatting with it, even when it misbehaved.	1	2	3	4	5	6

Is English your first language? Yes ☐ No ☐ I prefer not to say ☐

Your information and responses will remain anonymous and confidential. The data collected will be only be reported as a collective combined total and only the researcher will have knowledge of your individual answers.

☒

Appendix D

Chatbot templates

Below are some sample *templates* used by the chatbot, including some of the modifications made to incorporate the SUPER sample bot [ALICE AI Foundation, n.d.] to the shoping mall domain used in this project.

```
<category><pattern>*</pattern>
<template><srai>UDC</srai></template>
</category>
<category><pattern>UDC</pattern>
<template><random>
<li><srai>RANDOM PICKUP LINE</srai></li>
<li><srai>INQUIRY AGE</srai></li>
<li><srai>INQUIRY LOCATION</srai></li>
</random></template>
</category>
<category><pattern>RANDOM BOT INITIATION</pattern>
<template><think><set var="name"><get name="name"/></set></think>
<condition var="name">
<li value="unknown">Hello there!  What is your name?</li>
<li><srai>BEGIN NEW TOPIC</srai></li>
</condition></template>
</category>
<category><pattern>RANDOM PICKUP LINE</pattern>
<template><random>
<li>I am not sure what you mean.</li>
<li>Could you please rephrase that?</li>
<li>I am sorry. I do not know what you mean by that.</li>
<li>I might need some further explanation on what you require.</li>
<li>Could you please be more specific?</li>
<li>I am sorry <get name="name"/>. I am afraid I can not do that.</li>
</random></template>
</category>
```

```
<category><pattern>BEGIN NEW TOPIC</pattern>
<template>what would you like to talk about?</template>
</category>
<category><pattern>^ WHAT CAN * DO AROUND ^</pattern>
<template><srai>INQUIRY ACTIVITIES</srai></template>
</category>
<category><pattern>^ WHAT IS THERE AROUND ^</pattern>
<template><srai>INQUIRY ACTIVITIES</srai></template>
</category>
<category><pattern>WHAT IS THERE TO DO AROUND ^</pattern>
<template><srai>INQUIRY ACTIVITIES</srai></template>
</category>
<category><pattern>WHAT TIME * MALL CLOSES</pattern>
<template><srai>MALL CLOSING TIME</srai></template>
</category>
<category><pattern>WHEN * THE MALL CLOSES</pattern>
<template><srai>MALL CLOSING TIME</srai></template>
</category>
<category><pattern>WHEN * THE MALL CLOSING</pattern>
<template><srai>MALL CLOSING TIME</srai></template>
</category>
<category><pattern>MALL CLOSING TIME</pattern>
<template>The mall closes at 8 pm</template>
</category>
<category><pattern>* BATHROOM *</pattern>
<template>All bathrooms are located on my left, second door on the right</template>
</category>

<category><pattern>INQUIRY ACTIVITIES</pattern>
<template><random><li>There are a lot of things to do around here. You can drink
a coffee or have something to eat, there are numerous shops to do your shopping,
there is a cinema. Or try talking to me!</li>
<li>Quite a lot of things.</li></random></template>
</category>
```

Bibliography

Al Moubayed, S., Beskow, J., Skantze, G. and Granström, B. [2012], 'Furhat: A back-projected human-like robot head for multiparty human-machine interaction', *Cognitive Behavioural Systems* pp. 114–130.

ALICE AI Foundation [2013], 'Program AB'.
URL: *https://code.google.com/archive/p/program-ab/*

ALICE AI Foundation [n.d.], 'S.u.p.e.r. aiml bot'.
URL: *https://code.google.com/archive/p/aiml-en-us-foundation-super/*

API, M. S. [2016], 'Microsoft speech api (sapi) 5.4'.
URL: *https://msdn.microsoft.com/en-us/library/ee125663(v=vs.85).aspx*

Athanaselis, T., Bakamidis, S. and Dologlou, I. [2007], Evaluating ASR confidence score performance in different noisy conditions, *in* 'Proceedings of the 12-th International Conference Speech and Computer (SPECOM)', Moscow, pp. 207–211.

Bhatnagar, ShalabhLakshmanan, K. [2015], 'Multiscale q-learning with linear function approximation', *Discrete Event Dynamic Systems* **26**(3), 477–509.

Block, N. [1981], 'Psychologism and behaviorism', *The Philosophical Review* **90**(1), 5.

Byford, S. [2014], 'Softbank announces emotional robots to staff its stores and watch your baby'.
URL: *http://www.theverge.com/2014/6/5/5781628/softbank-announces-pepper-robot*

Chakrabarti, C. and Luger, G. F. [2015], 'Artificial conversations for customer service chatter bots: Architecture, algorithms, and evaluation metrics', *Expert Systems with Applications* **42**(20), 6878–6897.

Cheongjae, L., Sangkeun, J., Kyungduk, K., Donghyeon, L. and Gary Geunbae, L. [2010], 'Recent approaches to dialog management for spoken dialog systems', *Journal of Computing Science and Engineering* **4**(1).

Deemter, K. v., Theune, M. and Krahmer, E. [2005], 'Real versus template-based natural language generation: A false opposition?', *Computational Linguistics* **31**(1), 15–24.

Developer.nuance.com [2016].
URL: *https://developer.nuance.com/public/index.php?task=home*

Dingli, A. and Scerri, D. [2013], 'Building a hybrid: Chatterbot - dialog system', *Text, Speech, and Dialogue* pp. 145–152.

EMOTE-Project [n.d.], *Perception module architecture.*
URL: *http://gaips.inesc-id.pt/emote/perception-module/*

Even-Dar, E. [2001], *Learning rates for Q-learning.*

Ferguson, G., Allen, J. F., Miller, B. W. and Ringger, E. K. [1996], The design and implementation of the trains-96 system: A prototype mixed-initiative planning assistant, Technical report.

Ferguson, GeorgeAllen, J. F. [1999], 'The rochester interactive planning system', *AAAI-99 Proceedings* .

Iristk.net-gestures [2016].
URL: *http://www.iristk.net/gestures.html*

Iristk.net-overview [2016].
URL: *http://www.iristk.net/overview.html*

Levin, E., Pieraccini, R. and Eckert, W. [2000], 'A stochastic model of human-machine interaction for learning dialog strategies', *IEEE Transactions on Speech and Audio Processing* **8**(1), 11–23.

Lison, P. [2015], 'A hybrid approach to dialogue management based on probabilistic rules', *Computer Speech & Language* **34**(1), 232–255.

MacGlashan, J. [2016], 'Burlap'.
URL: *http://burlap.cs.brown.edu/*

McTear, M. F. [2002], 'Spoken dialogue technology: enabling the conversational user interface', *CSUR* **34**(1), 90–169.

Mori, M. [2010].
URL: *http://spectrum.ieee.org/automaton/robotics/humanoids/the-uncanny-valley*

Nuance [2016], 'Nuance developers'.
URL: *https://developer.nuance.com/public/index.php?task=home*

Pieraccini, RobertoHuerta, J. [2005], 'Where do we go from here? research and commercial spoken dialog systems', *Proceedings of the 6th SIGdial Workshop on Discourse and Dialogue* .

Platform, G. C. [2016], 'Speech api - speech recognition'.
URL: *https://cloud.google.com/speech/*

Reiter, E. and Dale, R. [1997], 'Building applied natural language generation systems', *Natural Language Engineering* **3**(1), 57–87.

Rieser, V. and Lemon, O. [2008], *Simulation-based Learning of Optimal Multimodal Presentation Strategies from Wizard-of-Oz data.*

Rieser, V. and Lemon, O. [2011], *Reinforcement learning for adaptive dialogue systems*, Springer.

Sato, T. and Hosokawa, K. [2012], 'Mona lisa effect of eyes and face', *i-Perception* **3**(9), 707–707.

Schumaker, R. P., Ginsburg, M., Chen, H. and Liu, Y. [2007], 'An evaluation of the chat and knowledge delivery components of a low-level dialog system:the az-alice experiment', *Decision Support Systems* **42**(4), 2236–2246.

Shawar, B. A. and Atwell, E. [2009], *Arabic question-answering via instance based learning from an FAQ corpus*, UCREL, Lancaster University.
URL: *http://eprints.whiterose.ac.uk/id/eprint/82302*

Shieber, S. M. [2006], *Does the Turing Test Demonstrate Intelligence or Not.*

Skantze, G. and Al Moubayed, S. [2012], 'Iristk: a statechart-based toolkit for multiparty face-to-face interaction', *Proceedings of the 14th ACM international conference on Multimodal interaction - ICMI '12* .

Skantze, G., Hjalmarsson, A. and Oertel, C. [2014], 'Turn-taking, feedback and joint attention in situated human–robot interaction', *Speech Communication* **65**, 50–66.

Sutton, Richard SBarto, A. G. [1998], *Reinforcement learning*, MIT Press.

Traum, D. R. [1996], 'Conversational agency: The trains-93 dialogue manager', *proceedings of the Twente Workshop on Language Technology 11:Dialogue Management in Natural Language Systems* pp. 1–11.

TURING, A. M. [1950], 'I.—computing machinery and intelligence', *Mind* **LIX**(236), 433–460.

van Woudenberg, A. [2014], A Chatbot Dialogue Manager, PhD thesis, Open University of the Netherlands.

Vinyals, O. and Le, Q. [2015], 'A Neural Conversational Model', *ArXiv e-prints* .

Waibel, A. [2004], 'Speech translation: Past, present and future', *INTERSPEECH-2004* p. 353–356.

Wallace, R. S. [2009], 'The anatomy of a.l.i.c.e.', *Parsing the Turing Test* pp. 181–210.

Weizenbaum, J. [1966], 'Eliza—a computer program for the study of natural language communication between man and machine', *Communications of the ACM* **9**(1), 36–45.

Williams, J. D. and Young, S. [2004], 'Using wizard-of-oz simulations to bootstrap reinforcement-learning-based dialog management systems'.

Xu, X., Zuo, L. and Huang, Z. [2014], 'Reinforcement learning algorithms with function approximation: Recent advances and applications', *Information Sciences* **261**, 1–31.

Young, S., Gasic, M., Thomson, B. and Williams, J. D. [2013], 'Pomdp-based statistical spoken dialog systems: A review', *Proceedings of the IEEE* **101**(5), 1160–1179.

Zadrozny, W., Budzikowska, M., Chai, J., Kambhatla, N., Levesque, S. and Nicolov, N. [2000], 'Natural language dialogue for personalized interaction', *Communications of the ACM* **43**(8), 116–120.