Human-Humanoid Interaction by Verbal Dialogue
(Nao Humanoid Robot with a Dialogue Toolkit)
Abstract

This master thesis describes a work that develop a spoken natural language user interface for humanoid robot. There have been many attempts and studies to develop human robot interaction techniques. However, it is still complicated for us to create a spoken dialogue system for the humanoid robot. In order to solve this problem, we propose to use a natural language user interface using a linguistic approach. In this approach we used the Yet Another Dialogue Tool Kit (YADTK), which is a rule-based spoken dialogue system. This method gives a flexibility to make humanoid robot communicate naturally as a human, and become a great intelligent personal assistance. By linking a dialogue toolkit with humanoid robot software, it is possible to control and interact verbally. The dialogue toolkit is applied on a Nao humanoid robot. Understanding the abilities of the Nao robot has shown a powerful demonstration with good results. The successful results of this technique elicit an important aspect that we can apply to any humanoid robot in order to extend its capabilities in terms of verbal interaction. As verbal interaction is a key feature of humans, this can also be extended to artificial intelligence of robot.

In this thesis, we will show how dialogue toolkit (YADTK) works and how the Nao robot is controlled by YADTK. Then extending abilities of Nao we will use vision to explore environment for navigation. So the Nao robot get an ability to explore an environment with verbal aid from human user. That will make Nao robot socially assistive and more interactive in terms of self-consciousness.

**Keywords:** artificial intelligence, dialogue toolkit, humanoid robot, intelligent personal assistance, linguistic, natural language user interface
This page is intentionally left blank for printing purpose.
Acknowledgements

The present thesis is a master thesis work of half year, where many ideas were sketched during master program course studies at Ecole Centrale de Nantes (ECN), Nantes, France. The work is conducted as an intern at the Institut de Recherche en Communications et Cybernétique de Nantes (IRCCyN) laboratory in the Humanoid Robotics Department (from Feb. 2015 until now). Needless to say that this project couldn’t have been completed without the help and support of many people that accompanied me on this project.

My first thanks go to my thesis supervisor: Yannick Aoustin and Jérôme Lehuen, I personally appreciate their guidance and encouragements on my work which have been invaluable. Their contribution with numerous advices on several revisions of this dissertation played a key role in helping me improve the form and content of the final draft. I am particularly grateful of Jérôme for allowing me to use and understand the dialogue toolkit, Yet Another Dialogue Tool Kit (YADTK), developed by him and allowed me to contribute to the development of the dialogue toolkit for the humanoid robots, and also for accepting to supervise a thesis topic which ventured outside the group’s traditional areas of expertise that made me a contributor to Language and Speech Technology (LST), University of Maine, Le Mans, France. I am equally thankful to Yannick, who always inspired me for work, and gave many guidance and opportunities to present the research work at several conferences.

It is a fortune to thank my colleagues for the great working environment and friendly atmosphere during all thesis work. I especially thank to all my IRCCyN workmates for making me feel welcome and part of this fantastic work environment from day one. This is an opportunity to equally thank all the staff of the Master in Automatique Robotique et Informatique Applique (ARIA) program for their indirect contribution. The first year program chair Prof. Wisama Khalil, I am thankful to him for his great support and I had great opportunity to learn robotics under his guidance. I also appreciate the second year program chair Prof. Phillipe Martinet, for his continuous support and guidance throughout the master program.
I consider my privilege to thank Erasmus Munsdus scholarship program that support my whole master program including thesis work. I especially, thanks to India4EU project under the framework of Erasmus Mundus program for great support both personally and academically.

The thesis work was appreciated by the chief creative officer of Aldebaran, Jérôme Monceaux, I am thankful to him for inviting at Artificial Intelligence Lab, Aldebaran, Paris and gave me an opportunity to explore and understand current research work on interaction of robots, Nao and Pepper robot, of the company.

Last but not the least, I am wordless to thank my life, parent and friends, for their love and support throughout the academics. Thank you for everything.

Chandrakant Bothe

Nantes, 20th July 2015
# Contents

Abstract .......................................................................................................................... III
List of Figures .................................................................................................................. IX
List of Tables ................................................................................................................... XI
List of Abbreviations ....................................................................................................... XIII

I Introduction .................................................................................................................... 1
  I.1 State of the Art ......................................................................................................... 2
  I.2 Literature Survey .................................................................................................... 3
  I.3 Motivation ................................................................................................................ 6
  I.4 Outline of the Thesis ............................................................................................... 7

II Natural Language Processing: Background ................................................................ 9
  II.1 What is Language? .................................................................................................. 9
  II.2 Natural Language Processing ............................................................................... 10
  II.3 Human-Humanoid Interaction ............................................................................. 12
  II.4 Multi-modal Systems ............................................................................................ 13
  II.5 Summary ............................................................................................................... 13

III Platforms: Nao and YADTK .................................................................................... 15
  III.1 Nao Humanoid Robot .......................................................................................... 16
    III.1.1 Features of Nao Humanoid Robot .................................................................. 16
    III.1.2 Programming the Robot ............................................................................... 17
    III.1.3 Robot’s Inbuilt Dialogue Module: ALDiaolog .............................................. 19
  III.2 YADTK: A Dialogue Toolkit ................................................................................ 20
    III.2.1 The Granule Model ....................................................................................... 22
    III.2.2 The YASPML Formalism ............................................................................. 26
      III.2.2.1 Methods of the YASPML Formalism .................................................... 27
    III.2.3 The YASP Understanding Module ............................................................... 32
      III.2.3.1 The Abilities of YASP Understanding Module ....................................... 34
    III.2.4 The YAGE Generation Module .................................................................... 37
    III.2.5 The YADE Dialogue Control Module ......................................................... 38
    III.2.6 The YADEML Formalism ............................................................................ 39
III.3 Summary .......................................................... 43

IV Integration of Nao with YADTK .................................................. 45

IV.1 Software Architecture ......................................................... 45
IV.2 Built-in Mechanism to Control the Robot .................................... 47
IV.3 New Mechanism to Control the Robot ........................................ 53
IV.4 Methods to Call Functions on Server/Client Side ............................. 56
  IV.4.1 Pre-executing Functions on Server Side .................................... 57
  IV.2.2 Post-executing Functions on Client Side .................................... 59
  IV.2.3 Concurrent Execution of the Functions on both sides .................... 62
IV.3 Queueing Dialogues ................................................................ 63
  IV.3.1 Queueing Dialogues: Time Based ........................................... 63
  IV.3.2 Queueing Dialogues: Sensor Based ......................................... 64
IV.4 Multiprocessing Functionality .................................................... 69
IV.5 Summary ............................................................................. 71

V Navigating Nao with Verbal Aid: .................................................. 73

V.1 Description ........................................................................... 73
V.2 Generating the QR Code ............................................................ 74
V.3 Reading the QR Code ............................................................... 76
V.4 Navigation by Reading QR Code ................................................ 78
  V.4.1 Searching QR Code while Rotating the Head ............................. 78
  V.4.2 Moving Towards the found QR Code ...................................... 82
V.5 Summary ............................................................................. 84

VI Concluding Remarks ................................................................. 85

VI.1 Conclusions .......................................................................... 85
VI.2 Future Scope .......................................................................... 86

Bibliography ............................................................................... 87

Appendices .................................................................................. 89
List of Figures

Figure 1 General Spoken Dialogue System ................................................................. 11
Figure 2 General Architecture of Nao Verbal Interactive System ............................. 15
Figure 3 Nao Hardware Overview (Aldebaran’s Nao) ................................................ 16
Figure 4 YADTK Client-Server Architecture ............................................................ 20
Figure 5 YADTK Simplified Client-Server Architecture ........................................... 21
Figure 6 The Granule Model ..................................................................................... 23
Figure 7 Granule Structure ....................................................................................... 25
Figure 8 Flexible Dependency of Granule ............................................................... 28
Figure 9 Multiple Dependency in Granules ............................................................. 29
Figure 10 Boolean Constraints in Granule ............................................................... 30
Figure 11 Interrogative and Negative Forms ............................................................ 31
Figure 12 Granule Structure of a Complex Utterance .............................................. 33
Figure 13 Dealing with redundancies: Merging three G-structures ....................... 34
Figure 14 Example of Dealing with ambiguities ...................................................... 35
Figure 15 Example of a “Granule Guessing” ............................................................ 36
Figure 16 Example of two firing contexts ............................................................... 38
Figure 17 Basic Software Architecture of Nao with YADTK .................................. 46
Figure 18 Complete Software Architecture of Nao with YADTK ............................. 47
Figure 19 Generated QR code for data (a) “First Wall” and (b) “Second Wall” ........ 75
Figure 20 Training the robot to read QR codes from the robot camera .................. 78
Figure 21 Head Directions for searching the object .................................................. 79
List of Tables

Table 1 iCub Specific Action Commands.................................................................................. 4
Table 2 Specifications of the robot ............................................................................................... 17
Table 3 Tabular representation of Granule elements.................................................................. 23
Table 4 YASPML formalism syntax .............................................................................................. 27
Table 5 Tabular representation of example of two firing context.................................................. 39
This page is intentionally left blank for printing purpose.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>ASR</td>
<td>Automatic Speech Recognition</td>
</tr>
<tr>
<td>CLIPS</td>
<td>C Language Integrated Production System</td>
</tr>
<tr>
<td>CSLU</td>
<td>Center for Spoken Language Understanding</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HCI</td>
<td>Human-Computer Interaction</td>
</tr>
<tr>
<td>HRI</td>
<td>Human-Robot Interaction</td>
</tr>
<tr>
<td>NLI</td>
<td>Natural Language Interface</td>
</tr>
<tr>
<td>NLP</td>
<td>Natural Language Processing</td>
</tr>
<tr>
<td>NLSI</td>
<td>Natural Language Speech Interface</td>
</tr>
<tr>
<td>NLTI</td>
<td>Natural Language Text Interface</td>
</tr>
<tr>
<td>QR</td>
<td>Quick Response code</td>
</tr>
<tr>
<td>RAD</td>
<td>Rapid Application Development Toolkit</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>TCL</td>
<td>Tool Command Language</td>
</tr>
<tr>
<td>TTS</td>
<td>Text-To-Speech</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
This page is intentionally left blank for printing purpose.
Introduction

“The user should not have to become a programmer, or rely on a programmer, to alter the robot’s behaviour, and the user should not have to learn specialized technical vocabularies to request action from a robot.”

Crangle and Suppes (1994) [2]

A very large part of our waking hours is spent in social interactions via natural languages. There are thousands of languages in the world, and in our daily lives the spoken language plays a very important role. The spoken system is the one which has tremendous efficient ability to express our ideas clearly to others.

Technically speaking, after command line prompt, the graphical user interface (GUI) played a very important and adorable role in the interactions with the machines and computers. The time has changes and Natural Language User Interface is getting a complete attention of developer and society. In the next few years, the natural language interaction seems to find a lot of interest to develop a user-friendly spoken technologies.

As the technologies like mobile phone, personal computers, navigation systems etc. are gaining autonomy with sophistication, the importance of user interface increases. Today, the humanoid robots are very advanced, and still the importance remains at the end-user interface. After all, humanoid robots are made to look like us, act like us, so it is also important that they must interact and speak like us. Current research on human–humanoid interaction is precisely trying to realize these goals.

The robots are trained to speak like human, with rules and probabilistic approaches. In the rule based approach the robot has ‘knowledge’ in its database and if user asks him, he search in the knowledge and finds answer. For example, if we said “Hello, Robot!” the robot replies
"Hello, Sir", or depending on time, if it is morning robot can reply “Good morning, Sir.” This gave robot some basic rules for making artificial decisions, such as if it is morning ‘good morning’, or if evening time ‘good evening’ etc.

I.1 State of the Art

Today humanoid robots are able to work with humans in useful, appropriate and cooperative activities. The learning should take place in a natural and comfortable way for the user. This kind of learning can take place at the time of interaction, as the robot continuously acquires knowledge of the structure of the interaction, it can apply that knowledge in order to anticipate the behaviour of the user. This anticipation can be expressed both in terms of the actions performed by the robot, as well as by its style of verbal communication, anticipation is one of the hallmarks of intellect.

There is a large development of spoken language and made applicable to human-robot interaction and the user can easily understand what the robot learned by demonstration. Through exposure to repetition the robot should automatically extract and exploit the basic regularities given in Domeney (2008)[4], from human-robot interaction experiments, the architecture is characterized by the maintenance and use of an interaction history. This is a literal record of all past interactions or dialogue rules that have taken place by a robot with the user. The system continuously searches the interaction history for sequences whose onset matches the actions that are currently being interacted. If the robot is able to recognize such matches, will allow the robot to take different levels of predictions of activity, as predicted sequences are successively validated by the user, the level of anticipation and learning of the robot increases.

Crangle and Suppes (1994), [2], stated: “The user should not have to become a programmer, or rely on a programmer, to alter the robot’s behavior, and the user should not have to learn specialized technical vocabularies to request action from a robot.” Spoken language provides a very rich and direct means of communication between cooperating humans Pickering (2004), [24]. Language essentially provides a vector for the transmission of meaning between agents, and should thus be well adapted for allowing humans to transmit meaning to robots. This raises technical issues of how to extract meaning from language. Construction grammar provides a linguistic formalism for achieving the required link from language to meaning Goldberg (2003), [7]. Indeed, grammatical constructions define the direct mapping from sentences to meaning. The meaning of a sentence such as (1) is represented in a predicate-argument (PA) structure as in (2), based on generalized abstract structures as in (3). The power of these constructions is that they employ abstract argument “variables” that can take an open set of values.

(1) John, put the ball on the table.
(2) Transport (John, Ball, Table)
(3) Event (Agent, Object, Recipient)

We previously developed a system that generates PA representations (i.e. meanings) from
event sequences. When humans performed events and described what they were doing, the resulting <sentence, meaning> input pairs allowed a separate learning system to acquire a set of grammatical constructions defining the sentences. The resulting system could describe new events and answer questions with the resulting set of learning grammatical constructions. PA representations can be applied to commanding actions as well as describing them, Peter (2007), [5].

I.2 Literature Survey

The introduction of a humanoid robot system for human–robot communication is the key for the work carried out in the lab. In the work of Ido J. (2006) [10], in Japan, the humanoid robot used HRP-2 however the dialogue system uses a large vocabulary for continuous speech recognition. It is used as a research platform to develop an intelligent real-world interface using various information technologies. The functions implemented for human robot interaction are: speech recognition, voice synthesizing, facial information measurement, portrait drawing and gesture recognition, etc., Igor R. (2014) [11]. The most important to note that the speech recognition used a large vocabulary for the dialogue system. This leads to the user to utilize the same vocabularies that the robot knows. The user is bound to remember the technical vocabulary.

The multi-modal conversational interaction system on the Nao humanoid robot, the WikiTalk, was implemented, using an existing spoken dialogue system for open-domain conversations. This is a great extension for the robot’s interaction capabilities by enabling Nao to talk about many topics available on Wikipedia. In addition to verbal interaction, there was a wide range development of multimodal interactive behaviors by the robot including face tracking, nodding, communicative gesturing, proximity detection and tactile interruption. In Igor R. (2014), [11], two different ROS packages proposed to enrich the tele-operation of the robot NAO: speech-based tele-operation and gesture-based tele-operation together with arm control. These packages have been used and evaluated in a human mimicking experiment and the tools offered can serve as a base for many applications.

The application example is the ALIZ-E Project, this project is aimed to build the artificial intelligence (AI) for small social robots and to study how young people would respond to these robots. Long-term human–robot interaction has tremendous potential, as the robots can then be used to bond with people, which can in turn be used to provide support and education. As an application domain, the project focused on children with diabetes, whom the robots help by offering training and entertainment. The work on an event-based approach for integrating a conversational human and humanoid robot interaction system carried out by Ivana et al. (2011), [12], this approach has been instantiated using the Urbi middleware on a Nao robot, used as a test bed for investigating child–robot interaction in the ALIZ-E project. The main focus was on the implementation for two scenarios: an imitation game of arm movements and the quiz game for children.
The work presented by Domney et al. (2008), [6], is very impressive, the anticipation technique to predict the human activity, they used iCub humanoid robot developed by Italian Institute of Technology. It has been designed to represent the size of a three and a half years old child (approximately 1m tall) with large dexterity. The dialogue management and spoken language processing (voice recognition, and synthesis) is provided by the CSLU (Center for Spoken Language Understanding) Rapid Application Development (RAD) Toolkit. RAD provides a state based dialogue system capability, in which the passage from one state to another occurs as a function of recognition of spoken words or phrases; or evaluation of Boolean expressions.

Via the TCL (Tool Command Language) language one can open robot platform ports to the joints of the iCub robot. The robot is thus controlled via spoken language, via interaction with the different joints through the YARP (Yet Another Robot Platform) port interface. The behavioral result of a spoken action command that is issued either directly, or as part of a learned plan is the execution of the corresponding action on the robot. Based on the preliminary analysis of the table-building scenario, a set of primitive actions was identified for the iCub. Each of these actions, specified in Table 1, corresponds to a particular posture or posture sequence examples that is specified as the angles for a subset of the 53 DOFs. These actions have been implemented as vectors of joint angles for controlling the head, torso, and left and right arms.

Table 1 iCub Specific Action Commands

<table>
<thead>
<tr>
<th>Motor Command</th>
<th>Resulting Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reach</td>
<td>Position left hand next to closest table leg</td>
</tr>
<tr>
<td>Grasp</td>
<td>Close left hand</td>
</tr>
<tr>
<td>Lift</td>
<td>Raise left hand</td>
</tr>
<tr>
<td>Pass</td>
<td>Turn trunk and left shoulder towards user</td>
</tr>
<tr>
<td>Open</td>
<td>Open left hand</td>
</tr>
<tr>
<td>Hold</td>
<td>Bimanually coordinated holding</td>
</tr>
<tr>
<td>Release</td>
<td>Place both hands in upward safe position</td>
</tr>
<tr>
<td>Wait</td>
<td>Suspend until OK signal</td>
</tr>
</tbody>
</table>

In addition to allowing the user to issue iCub specific motion commands, the Spoken Language Programming system implemented the different levels of anticipation. The RAD programming environment provides a GUI with functional modules that implement speech recognition and synthesis, and flow of control based on recognition results and related logic.

As one of the long-term goals for humanoid robotics is to have these robots working side-by-side with humans, helping the humans in a variety of open ended tasks, which can
change in real-time. In such contexts a crucial component of the robot behavior will be to adapt as rapidly as possible to regularities that can be learned from the human. This will allow the robot to anticipate predictable events, in order to render the interaction more fluid. This will be particularly pertinent in the context of tasks that will be repeated several times, or that contain sub-tasks that will be repeated within the global task. Through exposure to repetition the robot should automatically extract and exploit the basic regularities. The results from human-robot cooperation experiments are in the context of a cooperative assembly task. The architecture is characterized by the maintenance and use of an “interaction history” a literal record of all past interactions that have taken place.

During on-line interaction, the system continuously searches the interaction history for sequences whose onset matches the actions that are currently being invoked. Recognition of such matches allows the robot to take different levels of anticipatory activity. As predicted sequences are successively validated by the user, the level of anticipation and learning increases. Level 1 anticipation allows the system to predict what the user will say, and thus eliminate the need for verification when the prediction holds. At Level 2 allows the system to take initiative to propose the predicted next event. At Level 3, the robot is highly confident and takes initiative to perform the predicted action. This demonstrates that how these progressive levels render the cooperative interaction more fluid and more rapid.

The development of the anticipation is impressive, but this is valid for small conversation of few words. As shown in the Table 1, the robot is able to follow the order from the user with predefined set of words that are assigned to specific tasks.

The recent system developed by Pointeau, G. (2014), [26], is capable of learning to extract the correct comprehension and production of personal pronouns and proper nouns during Human-Robot or Human-Human interactions. They use external 3D spatial and acoustic sensors with the robot iCub to allow the system to learn the proper mapping between different pronouns and names to their properties in different interaction contexts. The properties are Subject (Su), Speaker (Sp), Addressee (Ad) and Agent (Ag). A fast mapping system is used to extract correlation between the different properties. After a learning phase, the robot is able to find the missing property when only 3 out of 4 are known, or at least to discriminate which word cannot be used to be the lacking property.

As we can see from the above work, the implementation of the dialogue systems are quite similar. All the implementations uses a large corpora\textsuperscript{1} of vocabulary and grammar. The HRP-2, the dialogue system was developed for very simple task and the use the large data, like asking Office and laboratory locations, Extension telephone numbers of staffs or making greetings etc. only.

For the humanoid robot iCub, the implementation have simple tasks like Reach, Grasp, Lift, Pass, Open, etc. The system of dialogue communication is good, that they used different levels of the hierarchy, for anticipation, taking predictive decision and approving it from the human user, but at the same time the communication remains as a single word.

\textsuperscript{1} A corpus is a large and structured set of texts; plural: corpora
communication or ordering the robot by a single word. This cannot be considered as natural language interfacing.

In the case study of humanoid robot Nao, task made him to talk about several topics from the Wikipedia articles. The important to learn from this was the general approach to take that the robot should come pre-equipped with a set of capabilities, including grasping particular objects, moving to useful postures that allow it to pass objects to the user, take objects from the user, hold objects while the user works on them etc.

1.3 Motivation

Natural language interfaces are turning into a real-world interface convention. It is getting same interest like the Graphical User Interface (GUI), the GUI is overlapped and largely replaced the command line, and now the Natural Language Processing (NLP) is grabbing its place. NLP is being used by robots, the Internet of things, wearables, and especially conversational systems like Apple’s Siri, Google’s Now, Microsoft’s Cortana, Nuance’s Nina, Amazon’s Echo and others. These interfaces are designed to simplify, speed up, and improve task completion. Natural language interaction with robots, if anything, is an interface. It’s a form of user experience that requires design, Mark Stephen (2015), [16].

In our daily life we all use language as our key interface with other people. We all have many problems talking with one another because of diversity, so designing talking interfaces is a hard job. Topic management, dialogue turn-taking, segue management, association with iconic gestures, and thousands of other aspects of communication sit right in the middle of this emerging design discipline.

Traditionally the design, even the software design, employs the famous architectural adage of, “Form follows function.” But when it comes to natural language interfaces, and robotics in general, we enter a new kind of design in which function follows form. The form of human interaction becomes the function in making something metaphoric, simple, fun and useful.

When it comes to human-robot interaction, the form defines the function. There are at least three reasons for this, and all of them are psychological, for social robots in particular.

(1) Robots should look like us.
(2) Robots should speak like us.
(3) Robots should be polite.

From these basic reasons, it is clear that the robot should look like us, the human, there is no more choice than a humanoid robot. Finally we arrived at the step to know the humanoid robot, to bring them into the society, though, beside the very accurate and high grade sensors, walking is too hard to calculate by computational processing. If the robot falls, it takes a serious digger, damaging the sensors and body of the robot. Anyways, shape
of the robot doesn’t matter for the developers, but the social robots should have an attractive face, that is able to make some expressions like human, maybe mimic the human expression.

Just like the social robots are ensure to look like us, there also need to design robots that speak like us. Natural Language Processing technology is working well enough to complete end user tasks. Here the challenge is to design spoken/verbal interaction system, human-like interaction as the core metaphor of naturally interacting robots.

The politeness is almost a human input to the robots or any machine. Most user actually try too polite and cooperative to the systems (like humor, aggressiveness, and gender). In a survey it is found that the human becomes more polite with the machine than one might guess. For example, if we watch a people talk with Apple’s Siri, they use the words like “Please” or “Thank You” more often. This survey was continued with natural language text and voice interfaces, in both of them, the results concluded that the social rules can apply to media and computers can be social initiators.

This leads to have interactive humanoid robots. The humanoid robots, for example, the Aldebaran’s Pepper system is doing a great job in the Soft Bank stores that interact with the customers and people of all ages admired the work. The next challenge is to develop a NLP for a humanoid robot, because one must note that the Human-Computer Interaction (HCI) is different than Human-Robot Interaction (HRI). The difference between HCI and HRI is important to understand because of the dynamic and moving body of the robot.

1.4 Outline of the Thesis

So far we have seen the first chapter that introduced the importance of the user interfaces. The change in the technologies, considering sophistication and autonomy, the user interface remains most important factor to operate the systems. After command line as GUI taken the most attractive turn, at the same time, now a days the natural language interface is also becoming the most important media to control devices. For humanoid robot it is very important to interact verbally, if we want to make the robot like us.

State of the art shows the basic operation we do with the dialogues, moving to literature survey gives the existing work done in the field of Human-Humanoid Interaction. Then we have seen the motivation to make robot like us, make them look like us, speak like us and also robot must be polite.

A brief overview of the structure of thesis is provided below:

Chapter II: Natural Language Processing: Background

This chapter introduces the most important concepts of used in the field of natural language processing and the spoken dialogue systems. Starting with the review of basic concepts of
language and linguistics, exploring facts relevant to our work. After this natural linguistic overview, we move to multi-modality of the humanoid robots, emphasizing more on making the robot more interactive and expressive. Finally, we survey different methods and understand the rule-based spoken dialogue system.

**Chapter III: Platforms: Nao and YADTK**

As our main goal is to implement the Natural Language Interface both Natural Language Text Interface and Natural Language Speech Interface for the Humanoid Robot that lead to consider all the aspects of the development. While developing the human–humanoid interaction, it is expected that the robot have basic abilities to walk, turn, embedded vision, speech recognition, text-to-speech synthesis, etc. The humanoid robot Nao is one of the most affordable and efficient robot to develop natural language interface. We will see the general overview of the Nao robot with verbal interaction ability. This chapter is especially dedicated to understand the Nao humanoid robot and YADTK dialogue toolkit. Along with basic ideas of Nao robot controlled by Python SDK, there is a large explanation on the YADTK dialogue toolkit.

**Chapter IV: Integration of Nao with YADTK**

The most important part of the work is to integrate Nao's SDK with YADTK software module. Actually we will be controlling the robot by dialogue toolkit through text or speech. In this chapter we will see the developed software architecture, different methodologies to control the robot through dialogue toolkit. Then moving to the queueing techniques and multiprocessing abilities to make the robot more and more expressive.

**Chapter V: Navigating Nao with Verbal Aid: Task Implementation**

In this chapter, we will see the implementation of the particular task that will make the robot to explore the environment by using the vision. The corresponding sections will focus on how to generate QR code, how to read them, and then how to search them in the environment. On occurrence of the code asking the robot to follow any of them. This makes robot to navigate in the known environment with the verbal guidance of human user companion.

**Chapter VI: Concluding remarks**

In this chapter, we will conclude the thesis with extensive remarks and we will see some points that are not directly addressed in these chapters but has importance scope for the future work.
Natural Language Processing: Background

This chapter introduces the most important concepts that are used in the field of natural language processing and the spoken dialogue systems. We start by reviewing basic concepts of language and linguistics, exploring facts relevant to our work. A proper understanding of these components is essential for the design of spoken dialogue systems for robots. We survey different methods and understand the rule-based spoken dialogue system in natural language processing.

After this natural linguistic overview, we move to human-robot interaction and multimodality of the humanoid robots, emphasizing more on making the robot interactive and expressive.

II.1 What is Language?

The beginning of the 21st century marks a period, where humanoid robots and the study of human and artificial cognitive systems come in parallel to a level of social end-view. This is sufficient for significant progress to make these robots more human-like in their interactions. In this context, two domains of interaction that humans exploit with great fidelity are spoken language, and the visual ability to observe and understand intentional action. A good deal of research effort has been dedicated to the specification and implementation of spoken language systems for human-robot interaction, Peter F. (2008), [22].

Languages are sets of signs. Signs combine an exponent (a sequence of letters or sounds) with a meaning. Grammars are ways to generate signs from more basic signs. Signs combine
a form and a meaning, and they are identical with neither their exponent nor with their meaning, Markus Kracht (1989), [17], Petric F. et al. (2014), [23].

Language is a means to communicate, it is a semiotic system (the study of how meaning is created). By that we simply mean that it is a set of signs. Its sign is a pair consisting, in the words of Ferdinand de Saussure, of a signifier and a signified. We prefer to call the signifier the exponent and the signified the meaning. For example, in English the string ‘dog’ is a signifier, and its signified is, say, ‘doghood’ or the set of all dogs.

II.2 Natural Language Processing

Natural Language Processing (NLP) is a field of computer science, artificial intelligence, and computational linguistics concerned with the interactions between computers and human (natural) languages. As such, NLP is related to the area of human-computer interaction. Many challenges in NLP involve natural language understanding, that is, enabling computers to derive meaning from human or natural language input, and others involve natural language generation. In the context of developing the system we consider both interfaces text and speech, i.e. NLTI and NLSI.

There are many ways to understand the natural language, from the reference of the John Searle (1969) [15], book Speech Acts: An Essay in the Philosophy of Language, several speech acts taxonomy are given, which is mainly divided in five central categories:

**Assertives**: Committing the speaker to the truth of a proposition.

Examples: “I swear I saw him on the crime scene.”, “I bought more coffee.”

**Directives**: Attempts by the speaker to get the addressee to do something.

Examples: “Clean your room!”, “Could you post this for me?”

**Commissives**: Committing the speaker to some future course of action.

Examples: “I will deliver this review before Monday.”, “I promise to work on this.”

**Expressives**: Expressing the psychological state of the speaker about a state of affairs.

Examples: “I am so happy for you!”, “Apologies for being late.”

**Declaratives**: Bringing about a different state of the world by the utterance.

Examples: “You’re fired.”, “We decided to let you pass this exam.”

This list can be widen considering the modern taxonomy of dialogues acts, Pierre Lison (2013), [25]. Also one must understand that, each and every person has different habits, for example maybe one of the speech act say declarative “You’re fired” can also become a part of joke that really depends on the context the person is speaking. So understanding such ambiguities is sometimes difficult for us, then designing such intelligent natural language
processing becomes more and more challenging.

This becomes an important issue to program such understanding of language that considers not only the current dialogues but also in a way they are spoken, to understand the context the speaker want to convey his message. NLP research is gradually shifting from lexical semantics to compositional semantics and, further on, narrative understanding. Human-level natural language processing, however, is an AI-complete problem. That is, it is equivalent to solving the central artificial intelligence problem-making computers as intelligent as people, or strong AI.

NLP’s future is therefore tied closely to the development of AI in general. Natural language processing is the technology for dealing with our most ubiquitous product: human language. In the past decade, successful natural language processing applications have become part of our everyday experience, from spelling and grammar correction in word processors to machine translation on the web, from email spam detection to automatic question answering, from detecting people’s opinions about products or services to extracting appointments from your email.

The basic approach to build a spoken dialogue system, the components can be shown in the block diagram below in Figure 1:

As one can notice in the block diagram that the Natural Language Processing module includes Natural Language Understanding, Dialogue Manager and Natural Language Generation modules, that are responsible for processing the text and make the conversation logic programming.

There have been several approaches to build dialogue systems includes:

- Pattern matching based approach
- Finite state based approach
• Frame based approach
• Plan based approach

All of these are very popular and conveniently used to build depending on the level of standard of application. Sometimes, the developers uses more than one approach in the same dialogue system. This gives a flexibility to program in different ways, with the single platform. For example the dialogue toolkit used in this thesis “Yet Another Dialogue ToolKit” created to develop a rule-based spoken dialogue systems allow to program in many approaches seen above possibilities called a hybrid approach, Jérôme Lehuen (2015), [13]. This dialogue toolkit is explored in chapter III.2 YADTK: A Dialogue Toolkit.

II.3 Human-Humanoid Interaction

Human-Humanoid Interaction leads more towards making the robot social assistive, because the humanoid robots are made to look and speak like human, to mimic human behaviour and actions. This tends to see the robot not as a machine point of view but social creature point of view. The interaction component of a socially interactive robot can be classified by the following properties, David Feil-Seifer (2005), [3]:

1) Embodiment
2) Emotion
3) Dialog
4) Personality
5) Human-oriented perception
6) User modelling
7) Socially situated learning
8) Intentionality

Since the robots are computing-intensive systems designed to benefit humans, the HRI can be informed by the research in HCI. The HRI differs from HCI in four dimensions, which can be considered categories for a taxonomy of human-robot interaction Holly and Jill (2002), [9]. The four dimensions are (1) the levels of human interaction, (2) the necessity of environment interaction for mobile robots, (3) the dynamic nature of robots in their tendencies to develop hardware problems, and (4) the environment in which interactions occur, Scholtz J. (02), [28].

Considering these facts that the robot’s behaviour is dynamic, the general assumptions can be to presume the time factor, type and number of people interacting with robot, number of robots in the environment, the environment of interaction, etc.
II.4 Multi-modal Systems

The multimodal interaction\(^2\) provides the user with multiple modes of interacting with a system. A multimodal interface provides several distinct tools for input and output of data. For example, a multimodal question answering system employs multiple modalities (such as text and photo) at both question (input) and answer (output) level.

The multi-modality of the system is very important to make the robot more interactive. In the work presented, there is an attempt to make robot multi-modal. Several sensors are used alongside the dialogue system. The dialogues are fired on the occurrence of the events based on sensory feedback. The sensors used includes: tactile sensor, vision (camera), battery charge sensor, bumpers, internal heat sensor, etc. and many more.

II.5 Summary

In this chapter we have seen the most important concepts used in the field of natural language processing and the spoken dialogue systems. We understand what is language and linguistics. Then we properly understood the essential components for the design of spoken dialogue system. Then we studied human-humanoid interaction and multi-modality of the humanoid robots, emphasizing on making the robot more interactive and expressive.

\(^2\) Multimodal: definition from Wikipedia
III

Platforms: Nao and YADTK

The aim is to implement the Natural Language Interface both Natural Language Text Interface and Natural Language Speech Interface for the Humanoid Robot. Consideration of the affordability and efficiency is important. This leads to consider all the aspects of the development. While developing the human-humanoid interaction, it is expected that the robot have basic abilities to walk, turn, embedded vision, speech recognition, text-to-speech synthesis, etc. The humanoid robot Nao is one of the most affordable and efficient robot to develop natural language interface. General architecture of the Nao robot with verbal interaction ability can be very similar as seen in the Figure 1 General Spoken Dialogue System shown below in Figure 2:

![Figure 2 General Architecture of Nao Verbal Interactive System](image)

This chapter is dedicated to understand the Nao humanoid robot and YADTK dialogue toolkit. Along with basic ideas of Nao robot controlled by Python SDK, there is a large explanation on the YADTK dialogue toolkit taken from the documentation of the toolkit.
III.1 Nao Humanoid Robot

NAO is a 58cm tall humanoid robot, Aldebaran’s Nao, [1]. Nao is intended to be a friendly companion around the house. He moves, recognises faces, hears and most importantly talks! Since his birth in 2006, he has been constantly evolving to please, amuse and understand. Aldebaran created NAO to be a true daily companion. His humanoid form and extreme interactivity make him really endearing and loveable.

III.1.1 Features of Nao Humanoid Robot

The various versions of the Nao robotics platform feature either 14, 21 or 25 degrees of freedom (DoF). A specialised model with 21 DoF and no actuated hands was created for the Robocup competition. All Nao Academics versions feature an inertial measurement unit with accelerometer, Gyro-meter and four ultrasonic sensors that provide Nao with stability and positioning within space. The legged versions included eight force-sensing resistors and two bumpers. The most recent version of the robot, the 2014 Nao Evolution, features stronger metallic joints, improved grip and an enhanced sound source location system that utilises four directional microphones. Some details shown in Figure 3.

The Nao robot is controlled by a specialised Linux-based operating system, dubbed NAOqi. The OS powers the robot’s multimedia system, which includes four microphones (for voice recognition and sound localization), two speakers (for multilingual text-to-speech synthesis) and two HD cameras (for computer vision, including facial and shape recognition).

![Figure 3 Nao Hardware Overview](Aldebaran’s Nao)
The robot also comes with a software suite that includes a graphical programming tool ("Choregraphe"), simulation software and a software developer's kit (SDK). Nao is furthermore compatible with the Microsoft Robotics Studio, Cyberbotics Webots, and the Gostai URBI Studio. The specifications are shown in Table 2:

<table>
<thead>
<tr>
<th>Nao: Next Generation Humanoid Robot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height</td>
</tr>
<tr>
<td>Weight</td>
</tr>
<tr>
<td>Power supply</td>
</tr>
<tr>
<td>Autonomy</td>
</tr>
<tr>
<td>Degrees of freedom</td>
</tr>
<tr>
<td>CPU</td>
</tr>
<tr>
<td>Built-in OS</td>
</tr>
<tr>
<td>Compatible OS</td>
</tr>
<tr>
<td>Programming languages</td>
</tr>
<tr>
<td>Sensors</td>
</tr>
<tr>
<td>Connectivity</td>
</tr>
</tbody>
</table>

**III.1.2 Programming the Robot**

As we have seen that the robot Nao can be programmed either by using programming languages or graphical user interface software i.e. Choregraphe, Nao Documentation, [1]. In the context of using the humanoid robot with dialogue toolkit, the aim is to interface the robot controlling with the dialogue toolkit. Considering this fact, programming the robot in Python is important because the YADTK dialogue toolkit used here, its kernel is written in Python.

NAOqi Python SDK is used in this thesis work. The NAOqi, itself is the main Python module, which can be imported like: `import naoqi`. There are different objects: ALProxy, ALBroker, ALModule etc. these can be imported separately, e.g.

```python
from naoqi import ALProxy
```
The object ALProxy will be used often in the programming. This object is responsible to create proxies to the module for communication with the robot platform. Actually, the robot is remotely controlled by the computer system. Once the proxy is created the computer is able to send or receive the information to and from the robot. This creates a communication between the robot and the computer, provided that both have established Ethernet or Wi-Fi connection. In this project, the Wi-Fi is used as a first connectivity medium as robot is moving. Also, if the robot is asked to move in the environment, it is recommended to use Wi-Fi instead of the Ethernet wired connection for safety.

There are two different constructors supported by ALProxy object, depending on whether or not one have a Broker instance available.

ALProxy(name, ip, port)

- **name**: is the name of the module
- **ip**: is the IP of the broker in which the module is running
- **port**: is the port of the broker

ALProxy(name)

- **name**: is the name of the module

Generally, ALProxy(name, ip, port) constructor will be used often because of its simplicity, e.g.

```python
memoryProxy = ALProxy("ALMemory", robotIP, PORT)
motionProxy = ALProxy("ALMotion", robotIP, PORT)
postureProxy = ALProxy("ALRobotPosture", robotIP, PORT)
ttsProxy = ALProxy("ALTextToSpeech", robotIP, PORT)
```

The memoryProxy generates a constructor to read the memory from module “ALMemory” of the robot memory at the robotIP and Port, to the corresponding robot.

Similarly, motionProxy uses “ALMotion” and postureProxy uses “ALRobotPosture” modules and ttsProxy uses “ALTextToSpeech” module.

Making Nao speak:

```python
from naoqi import ALProxy
ttsProxy = ALProxy(“ALTextToSpeech”, “robotIP”, 9559)
ttsProxy.say(“Hello, world!”)
```

This is the simplest example to make with Nao work or speak (here) through NAOqi Python SDK. There are several commands or called APIs in the Python SDK of the ANOqi, that can be found in the documentation provided by Aldebaran.
III.1.3 Robot’s Inbuilt Dialogue Module: ALDialog

The ALDialog module that is provided in the NAOqi Python SDK framework, allows us to endow robot with conversational skills by using a list of “rules” written and categorized in an appropriate way. ALDialog uses a list of written rules in order to manage the flow of the conversation between the human and the robot.

These rules are of two types: User rules and Proposal rules

- A User rule links a specific user input to possible robot output.

  u: (Hello Nao how are you today) Hello human, I am fine thank you and you?

- A Proposal rule triggers a specific robot output without any user output beforehand.

  proposal: Have you seen that guy on the TV yesterday?
  u1: (yes) He was crazy, no?
  u1: (no) Really, I need to tell you.

This is the simple and straight forward rules available for the robot’s conversations. This can be easily programmed for the robot in the NAOqi Python SDK using the ALDialog module. Our aim is to make the robot more interactive than available. The available dialogue rules are limited by the queue to put dialogues in pipe for conversation. It does not allow the user to write long conversations and also not very good for long sentences. All these facilities are provided by the YADTK dialogue toolkit, and the conversation is brought to more natural level.

---

### III.2 YADTK: A Dialogue Toolkit

The Yet Another Dialogue Tool Kit (YADTK) is an open-source toolkit to develop rule-based spoken dialogue systems. It is developed by Prof. Jérôme Lehuen, from the Language and Speech Technology Team, University of Maine, Le Mans, France. It is composed of several modules, all based on a unified semantic model, and called as the Granule Model, Jérôme and Thierry (2010), [14]. As the objective is the development of spoken dialogue systems, the YADTK toolkit is totally rule-based. Thus, there is no need to create and to annotate big corpora before beginning to design dialogue systems. In contrast, the knowledge must be hand defined using two very understandable (and extensible) XML formalisms⁴. It is maintainable and reusable, if it is well designed, organized and documented. YADTK is distributed without any integrated ASR module (automatic speech recognition). However, it is easy to join any additional module, thanks to its client-server architecture, implemented as Python scripts. In addition, YADTK have an ASR client that uses the Google™ Speech Recognition API, documentation of YADTK, [13]. The Figure 4 below shows all the possibilities of communication in this toolkit:

---

⁴ XML (Extensible Markup Language) is a general-purpose markup language for encoding and exchanging documents, as specified by the World Wide Web Consortium (W3C), cf. [http://w3.org/xml](http://w3.org/xml).
The YADTK dialogue toolkit contains:

- An understanding module called **YASP** (Yet Another Semantic Parser);
- A generation module called **YAGE** (Yet Another Generator);
- A dialogue controller called **YADE** (Yet Another Dialogue Engine);
- A graphical visualization tool that show semantic granule structures;
- A test module to proceed to **NRT** (Non-Regression Tests) during the development;
- A YASP client that allows you to test the YASP server separately;
- A YAGE client that allows you to test the YAGE server separately;
- A YADE client that allows you to test a keyboarded dialogue system;
- An ASR client that allows you to test a real spoken dialogue system;
- A speech synthesis feature (only on MacOS platforms).

As the communication layer is implemented with Python sockets, thus it is very easy to modify or to enhance the architecture. In the Figure 4, the yellow boxes, client and servers, are Python programs, rectangle page style box are XML code e.g. YADEML and YASPML Defs., circles are XSLT coding, and cylindrical shapes are CLIPS codes.

The Figure 5 below shows simple possible architecture of the toolkit:

As discussed above, the knowledge is described using two quite simple XML formalism:

- **YASPML** that describe the knowledge for the understanding and the generation abilities;
- **YADEML** that describe the knowledge for the inference and the dialogue abilities.

In principle, no syntactic parsing or POS (Part-of-Speech) tagging or additional knowledge are required. However, YADTK includes a minimalist POS Tagger for French and English.
This allows one to specify lemmatized forms into the syntactic patterns, and also to take advantage of the Granule Guesser. Except for these optional features, all the required knowledge is concentrated into the YASPML and the YADEML descriptions.

In this section, all the related figures that contain granules structures are taken from the documentation of YADTK, [13], which are generated by the Granule Visualizer which is a part of the YADTK toolkit.

### III.2.1 The Granule Model

The Granule Model is both a generalization and an operationalization of the *Schéma Actanciel* (stemma) proposed by the French linguist Lucien Tesnière in the frame of the Structural Syntax. Basically, a stemma is a tree diagram, in which the verbal node takes a central place. It is capable of governing a number of arguments called actants (actor, agent, patient, and instrument) and circonstants (adverbial phrases of time, of place, of manner, etc.). As the Fillmore’s Case Grammar, one of its major interests is to take into account both syntax, semantics and pragmatics.

In the Granule Model, to generalize this definition, a granule represents a significant unit (object, action, speech act, grammatical word, etc.) which is relevant or useful to the target task and/or to the dialogue. Each granule has a concept identifier, some semantic features that characterizing its offers, and can have some expectations. This defines the potential dependencies between granules. Each one has also a set of syntactic patterns that describes all its possible verbalizations.

The following Figure 6 represents the model of a granule having a valence of two, that to say with two potential dependencies. Each of them has a set of expected features, a set of required features, and a set of rejected features:

---

5 Lemmatization: the process of grouping together the different inflected forms of a word so they can be analysed as a single item
A semantic feature is just a symbolic constant that contributes to categorize a granule. The kinds of features that we use greatly depend on what we want to categorize. In the following example, some concepts are defined using positive and negative features. Negative (rejected) features are only used in the definitions of dependencies, not offers. It is given in Table 3:

Table 3 Tabular representation of Granule elements

<table>
<thead>
<tr>
<th></th>
<th>Human</th>
<th>Adult</th>
<th>Male</th>
<th>Female</th>
</tr>
</thead>
<tbody>
<tr>
<td>Man</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Woman</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Boy</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Girl</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cat</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Kitten</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Starting from the table above, the set \{+human, –adult\} can accepts the granules [boy] and [girl].

It is important to keep in mind that the objective is not to build a universal ontology, but a dialogue system into a closed world. So be pragmatic: *don’t try to define concepts by using features that are not useful in regard to the target system!*

A dependency is a potential link to child granules. It is characterized by a set of required, optional, and rejected features. The strength of a dependency is calculated from the number of common features between the expectations of the dependency and the offers of the child. A dependency can also be characterized by a role that the child granule will assume within the structure. For example, in the phrase *"a ticket to Paris"*, Paris is categorized as a *station*, but assumes the role *destination*, due to its position in the pattern (cf. the granule [ticket] below).
A syntactic pattern is a kind of regular expression composed of words or lemmatized forms (terminal symbols), and of dependency identifiers (non-terminal symbols). It is also possible to specify metadata in order to characterize different ways of expressing the granule (morphology, modality, style and level of language, etc.). As for the features, the choice of the metadata is totally open.

The granules are described within a straightforward formalism called YASPML. The frame below contains the YASPML description of some granules:

```
<granule concept="request" offers="speech-act">
  <dependency code="A1" expected="object requestable"/>
  <syntax pattern="I want (A1)" metadata="level:1"/>
  <syntax pattern="I would like (A1)" metadata="level:2"/>
  <syntax pattern="can I have/obtain (A1)" metadata="level:3 mode:inter"/>
</granule>

<granule concept="ticket" offers="object requestable">
  <dependency code="A1" expected="quantity"/>
  <dependency code="A2" expected="ticket-property"/>
  <dependency code="A3" expected="station" role="departure"/>
  <dependency code="A4" expected="station" role="destination"/>
  <syntax pattern="(A1) (A2) ticket(s)"/>
  <syntax pattern="(A1) (A2) ticket(s) to A4"/>
  <syntax pattern="(A1) (A2) ticket(s) from A3 to A4"/>
  <syntax pattern="A1 A3 A4"/>
</granule>

<granule concept="roundtrip" offers="ticket-property">
  <syntax pattern="round trip"/>
</granule>

<granule concept="paris" offers="place city station">
  <syntax pattern="paris"/>
</granule>

<granule concept="number:1" offers="number quantity">
  <syntax pattern="a"/>
  <syntax pattern="1"/>
  <syntax pattern="one"/>
</granule>

<granule concept="hello" offers="politeness">
  <syntax pattern="hello"/>
  <syntax pattern="hi" metadata="style:informal"/>
</granule>

<granule concept="please" offers="politeness">
  <syntax pattern="please"/>
</granule>
```

Thanks to these granule definitions, YASP can build the following structure of granules (called G-structure), shown in Figure 7, starting from the utterance:

"Hi, can I have one round-trip ticket to Paris please?"
This can be also represented as:

```plaintext
[hello] [request [ticket [number:1] [roundtrip] [paris]]] [please].
```

If this knowledge is well-defined (i.e. consistent and coherent), it can then be used for both understanding and generation. Thus, the system must be able to generate what it can understand, and conversely. To take benefit of this feature, all the granules must have at least one syntactic patterns with a `gen` attribute set to TRUE. The generation and the rephrase (partial or not) of an existing structure of granules is the job the YAGE module.
### III.2.2 The YASPML Formalism

The YASPML (YASP Modelling Language) is an XML formalism that allow you to describe a grammar for a specific dialogue application. The aim is not to describe all the subtlety of a natural language, but to allow a dialogue system to deal with natural language, taking into account some aspects of its variability. YASPML is based on the Granule Model described above. This is the description of a standard granule:

```xml
<granule concept="concept-ident" offers="set-of-features">
  <dependency code="Ai"
    role="optional-role"
    expected="set-of-features" (one at least and/or...)
    required="set-of-features" (all of them and/or...)
    rejected="set-of-features"> (any of them)
    ...
  </dependency>
  <syntax pattern="regular-expression" metadata="set-of-metadata">
    ...
  </syntax>
  ...
</granule>
```

- The **concept-ident** can be a simple token, like request, or a composed ident (a list of tokens separated by colon characters), like number:1. This can be used by the YADE dialogue. For example, the value 1 can be extracted from number:1. This is also used to deal with dates. Use a composed ident only if you plan to extract parts of this ident.

- The **set-of-features** in the offer attribute contains the semantic features that describe the granule in regard to the task and to the dialogue. The name of the granule (or each parts of its composed name) will be automatically added to the offers. Thus, a name (or a part of name) can be used as a expected, required, or rejected feature. Every feature identifier must be separately declared like this:
  ```xml
  <feature name="..."/>
  ```

- Each dependency is identified by a code Ai but it can also be characterized by a role that the child granule will endorse within a G-structure, like explained above. Every role identifier must be separately declared like this:
  ```xml
  <role name="..."/>
  ```

- The **set-of-features** in the following attributes are used to enable the feature-matching between the granule and its potential children:
  - At least one feature is required to satisfy the expected attribute;
  - All the features are required to satisfy the required attribute;
Any of the rejected features must appear in the child's offers. In the general case, at least one of these attributes must be filling. In special cases, that can be substituted by a Boolean constraint.

- A *regular-expression* is composed of terminal terms and non-terminal terms. This defines a reversible, context-free, semantic grammar. You can also specifies alternatives, optional terms, and lemmatized forms:

<table>
<thead>
<tr>
<th>Types of terms</th>
<th>Syntax / Example</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Constant words</td>
<td>… can I have …</td>
<td></td>
</tr>
<tr>
<td>2 Optional words</td>
<td>… tv (set) …</td>
<td>Combinable with 5,6,7,8</td>
</tr>
<tr>
<td>3 Dependency IDREFs</td>
<td>… A1 ticket …</td>
<td></td>
</tr>
<tr>
<td>4 Optional IDREFs</td>
<td>… (A1) ticket …</td>
<td></td>
</tr>
<tr>
<td>5 Optional characters</td>
<td>… child (ren) …</td>
<td></td>
</tr>
<tr>
<td>6 Open termination</td>
<td>… child+ …</td>
<td></td>
</tr>
<tr>
<td>7 Word alternative</td>
<td>… tv/television …</td>
<td>Combinable with 5,6,8</td>
</tr>
<tr>
<td>8 Lemmatized form</td>
<td>… (vouloir) …</td>
<td>For French* and for English</td>
</tr>
</tbody>
</table>

*Note: The use of lemmatized forms in French is based on BDLex [25] that is a commercial resource.*

- It is also possible to specify metadata for each pattern, in order to characterize different ways of speaking (morphology, modality, style or level of language, etc.). Like the features and the roles, the metadata must be separately declared:

```xml
<metadata name="..."/>
```

### III.2.2.1 Methods of the YASPML Formalism

#### a) Flexible dependencies

The general method for parsing consists in combining syntactic constraints (based on pattern-matching) and semantic constraints (based on feature-matching). In certain cases, it could be useful to bypass the syntactic constraints. Thus, a free (unconnected) granule can be "saved" by using only the feature-matching with a close granule. This is the aim of flexible dependencies. If there are several applicants, the nearest one is chosen. The example shown below that implements a flexible dependency.
The granule definition [room] contains a flexible dependency A2 that allows to connect (to rescue) free granules of type room-feature, without requiring any syntactic structure. That's why A2 does not need to appear in a syntactic pattern.

The G-structure in Figure 8 represents the utterance: I would like a room with a TV. The granule [television] was rescued thanks to the flexible dependency of the [room] granule. This is represented by a dotted edge. The role constraint isn't necessary, but it provide a better understanding of the structure.
b) Multiple dependencies

The multiple dependencies are a generalization of the flexible dependencies. The aim is to allow several granules to be connected via only one dependency. This is useful in the case where you want to collect a list of granules (objects, properties, informations, etc.) but you don't know how many.

```
<dependency id="A2" expected="room-feature" role="constraint" mult="TRUE"/>
```

In the following G-structure in Figure 9, that represents the phrase *a room with a TV and a jacuzzi*, the two granules [television] and [jacuzzi] are both connected to the dependency A2 of the granule [room]:

![Figure 9 Multiple Dependency in Granules](image)

c) Boolean constraints

The `<constraint test="..."/>` lines allows you to specify boolean constraints by using the CLIPS [24] syntax. This is useful in order to define specific grammatical structures, like the coordinating conjunction "and" for example. In this specific case, the two granules must have at least one common feature.

```
<granule concept="addition" offerexpr="(intersection ?offersA1 ?offersA2)"
  <dependency id="A1"/>
  <dependency id="A2"/>
  <constraint test="(intersectp ?offersA1 ?offersA2)"/>
  <syntax pattern="A1 and A2" gen="TRUE"/>
</granule>
```

The two variables `?offersA1` and `?offersA2` are automatically defined when the dependencies A1 and A2 are verified. So, in this case, the element `<constraint test="..."/>` requires a non empty intersection between `?offersA1` and `?offersA2`. The G-structure in Figure 10 corresponds to the phrase *a room with a TV and a jacuzzi* where [television] and [jacuzzi] share the feature *room-feature*. This is an alternative to the representation in Figure 9.
On the contrary, the interpretation of the phrase *Paris and a ticket* doesn't produce any granule [addition], because the granules [paris] and [ticket] have no feature in common.

![Figure 10 Boolean Constraints in Granule](image)

d) Calculated offers

Any granule always offers its name, that’s the reason why *addition* is an offer of [addition] in figure (ABOVE). It is interesting to note that the feature can be copied from its children. This is the effect of the `offereqre="..."` attribute, that contains a CLIPS expression.

e) Transferred offers

It is also possible to transfer offers from one (or more) dependency to the granule-father, by simply adding the code of the dependency to the list of offers.

f) Interrogative and negative forms

The fastest way to deal with interrogative and negative utterances is to take advantage of the metadata. Thus, interrogation and negation are just a way of talking about a concept, but not a concept:
Here are the G-structures corresponding to the following utterances, and structure shown in Figure 11:

- I want this one
- I want this room
- Can I have this room?
- I don't want this room

![Figure 11 Interrogative and Negative Forms](image-url)
### III.2.3 The YASP Understanding Module

The YASP module builds hierarchical structures of granules, directly from the user inputs, without any pre-segmentation or POS tagging. The parsing algorithm is based on a double principle of pattern matching and feature matching. An efficient forward chaining inference engine controls everything. YASP builds and stores partial interpretations, even if they are in competition. Then, it applies a conflict resolution strategy based on the number of words caught by the syntactic patterns. Finally, it tries to link unattached structures, only on the basis of feature matching. In this case, such granules become rescued. The output of the YASP server is an XML structure where each instantiated granule has the following attributes:

In certain cases, YASP can infer some granules, even if no syntactic pattern is recognized. Such granules are called inferred.

- An attribute id that gives the unique identifier of the granule;
- An attribute name that gives the name of the granule;
- An attribute offers that contains the offered features of the granule;
- An attribute metadata that contains the metadata of the used pattern;
- An attribute text that gives the recognized segment in the utterance;
- Two attributes POS and end that give the position of the segment;
- An attribute score that gives the calculated score of the granule;
- A Boolean attribute root that indicates if the granule is a root;
- A Boolean attribute rescued that indicates if the granule was rescued;
- A Boolean attribute inferred that indicates if the granule was inferred;

If the granule is a child, the following attributes are significant:

- An attribute code that gives the identifier of the dependency;
- An attribute role that gives the given role of the child in the structure;
- An attribute retained that contains the retained features during the connection;

YASP is able to deal with rather complex sentences. For example, the structure of granules in the following Figure 12, represents the utterance:

"Hello, can I have a room for five people, two couples and one child, from the twentieth to the twenty first of february in Marseille, and the following two nights in Avignon?"

This is an important feature of YADTK. If robot can understand such complex sentences, then the communication will be too natural and flawless. For example, for a robot, if the sentence consists of orders, emotions, and information, and if robot can extract this data, that will be a great achievement in the field of human-robot interaction.
Considering this result, we can see that the substructure `[duration:following [number:2] [time-unit:night]]` resulting from the phrase "the following two nights" can be substituted with the structure `[date-period [date [ordinal:22] [month:2]] [date [ordinal:23] [month:2]]]`. This kind of inference on structures of granules is fully implemented and integrated into the YADE module.

Now let’s see some abilities of YADTK to deal with redundancies, ambiguities and some features like The Granule Guesser and Non-Regression Tests.
**III.2.3.1 The Abilities of YASP Understanding Module**

**a) Dealing with redundancies**

To some extent, YASP can absorb redundancies in an utterance. This is possible thanks to a merging mechanism which searches for redundancies into the G-structures that come from the YASP parser.

For example, the utterance: "I want ... I want a ticket to Paris ... yes, that's it, a ticket to Paris" is firstly represented as shown on the left in Figure 13, then, the structures are merged as shown on the right. As the merging algorithm tries to keep the older granules, the granule [request]#1 was preserved during the merging of [request]#1 and [request]#17. Here are the trace of the inferences and the graphical representation. The deleted granules are drawn in grey:

### Merging of [ticket]#15 and [ticket]#16 (the latter is preserved)
<== [ticket]#15
<== [number:1]#6
<== [paris]#8

### Merging of [request]#17 and [request]#1 (the latter is preserved)
<== [request]#17

---

*Figure 13 Dealing with redundancies: Merging three G-structures*
b) Dealing with ambiguities

YASP is able to solve some ambiguities, if the context permits. For example in French, the word AVOCAT is an avocado or a lawyer. So the utterance “Je veux un avocat.” means “I want an avocado” or “I want a lawyer”. But if you say “Je veux un avocat compétent”, it is clear that you want a (competent) lawyer. Similarly, if you say “Je veux un avocat bien mûr”, it is clear that you want a (well ripe) avocado. The G-structure in Figure 14 illustrates this example of dealing with ambiguities.

![Figure 14 Example of Dealing with ambiguities](image)

c) The Granule Guesser

For certain applications, like information searching in open domain, we may need to identify some parts of utterances without having corresponding lexical data. This is the role of the Granule Guesser. For example, the phrase "the color of the black snow" can be interpreted without any granule definition for "the black snow". Indeed, the Granule Guesser can identify "the black snow" as a potential actant A2 of the granule [attribution], as long as there is enough information:

- A POS tagging that can identify "the black snow" as a nominal phrase (NP);
- A pattern for [attribution] in which it is specified that the A2 actant can be guessed:

```xml
<granule concept="color" offers="property">
  <syntax pattern="color"/>
</granule>

<granule concept="attribution">
  <dependency code="A1" expected="property"/>
  <dependency code="A2" expected="object" tag="NP"/>
  <syntax pattern="the A1 of A2"/>
</granule>

<granule concept="definition">
  <dependency code="A1" expected="attribution"/>
  <dependency code="A2" expected="definition" tag="NP"/>
  <syntax pattern="A1 is A2" metadata="mode:inter"/>
</granule>
```

In the following example, neither "the black snow" nor "an oxymoron" are known by YASP. However, the utterance "is the color of the black snow an oxymoron?" is well-interpreted, thanks to the Granule Guesser and the preceding YASP definitions. The guessed Granules are represented with a dotted line in the Figure 15:

```
is  the  color  of  the  black  snow  an  oxymoron
1  2  3  4  5  6  7  8  9
```

*Figure 15 Example of a "Granule Guessing"*
d) Non-Regression Testing

A classic problem that can occur while building rule-based grammars is the regression. Non-regression tests (NRT) are performed to test that adding or changing a rule has had the desired effect, and above all, to check that what worked before still works. In order to avoid regression, YADTK comes with NRT scripts. The basic methodology to proceed to NRT is the following:

1. Take an utterance from your corpus;
2. Submit it to YASP and examine the produced structure;
3. Modify the knowledge to produce a satisfactory structure (repeat step 2);
4. Add the XML structure to the test corpus tests.xml into the right data folder;
5. Start the NRT script: _testsYASP.command
6. If it’s OK, go to step 1;
7. Otherwise, repeat step 3.

If there is any error in granule, the script stops and will show error in the command window as GRANULE_ERROR or if a structure error occurs it shows STRUCTURE_ERROR.

e) Batched Parsing

The YASP Client for batched parsing is useful to parse big corpora with a YASPML grammar.

III.2.4 The YAGE Generation Module

As said previously, the YAGE generation module can generate a character string from a granule structure already present in the YADE working memory, or described as an XML literal description. To be sure that the system is able to generate all what it can understand, all the granules must have at least one syntactic patterns with a gen attribute set to TRUE. The patterns that you want be used for generation must have neither alternatives terms, like a/an nor optional character, like ticket(s). However, it is possible to use optional terms, like (very) or (A1).

The generation algorithm uses metadata in order to select the most suitable pattern. The principle is to maximize the length of the pattern and the number of common metadata between a granule and its sons (local constraints). It is also possible to influence the generation process by specifying global constraints. The latter are also specified by metadata. Thus, it is possible to set a dialogue style, as long as there are enough patterns for generation.
III.2.5 The YADE Dialogue Control Module

As goal is to propose a highly programmable tool, the choice was to minimize the number of built-in dialogue strategies, in order to allow programmers to develop their own strategies. In contrast, the YADTK is providing a rule-based language called YADEML, to deal with the G-structures produced by the YASP semantic parser. Actually, YADEML is used to describe inference rules and dialogue rules. What differentiates them is mainly situated in their consequences (actions part), that contain or not an answer of the system.

The dialogue controller plays a central role in a dialogue system. In particular, it is responsible of maintaining the coherence (locally and overall) into the conversation. Dialogue controllers are often based on a waiting principle that contributes to maintain the coherence:

- Between the statements and the current task (global contexts);
- Between the initiative and the reactive statements (local contexts).

Several approaches can be developed to implement a waiting principle. Some controllers are based on a finite state machine. This provides robustness, but also too little flexibility in dialogue. Other controllers are based on explicit waiting, which requires to define a declarative waiting system.

In summary, this is the typology of YADE rules, Table 5 gives detail and Figure 16 shows flow of the two firing context switching:

- SR (standalone rules): can be fired anywhere;
- IR (initiative rules): can be fired anywhere + open a new context;
- NR (nested rules): can be fired only in a specific context;
- NTR (nested terminal rules): can be fired only in a specific context + close the current context;
- NIR (nested initiative rules): can be fired only in a specific context + open a new context.

![Figure 16 Example of two firing contexts](image-url)
### III.2.6 The YADEML Formalism

Typically, the LHS (left-hand-side) of a YADEML rule describes a structure of granules within an XML formalism. If this description matches a substructure of granules inside the working memory, the RHS (right-hand-side) can be triggered. This is the general structure of a standard YADEML rule:

```xml
<rule descr="..." example="...">
  <conditions>
    ...
  </conditions>
  <actions>
    ...
  </actions>
  Nested rules can be written here...
</rule>
```

A YADEML rule do not have any name. The descr attribute contains a simple description (that will appear in the traces) and the example attribute just allow you to write an example of utterance that fires the rule. The `<conditions>... </conditions>` part contains the preconditions of the rule. Generally, a precondition is a XML literal description of a structure of granules that must match with the G-structures into the working memory.

For example, the following condition matches with a granule [date] that has a [calendar] child but no [month] child. The [date] granule is identified by a variable that can be used into the RHS:

<table>
<thead>
<tr>
<th></th>
<th>can fire anywhere</th>
<th>opens a new context</th>
<th>closes the current context</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>IR</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>NIR</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>NTR</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>NR</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
It is possible to filter a granule according to its name, its offers, its code, its role, its metadata, and if it's a root or not. It also possible to use variables into a granule's name, and to verify some Boolean constraints into the LHS:

```
<conditions>
  <granule concept="date" ident="?id">
    <granule concept="calendar"/>
    <no-granule concept="month"/>
  </granule>
</conditions>
```

Here, the date-error Boolean function is defined by the user, with a CLIPS syntax:

```
deffunction MAIN::date-error (?day ?month)
  (bind ?day (toNumber ?day))
  (bind ?month (toNumber ?month))
  (or
    (and (= ?month 1) (> ?day 31))
    (and (= ?month 2) (> ?day 28))
    (and (= ?month 3) (> ?day 31))
    (and (= ?month 4) (> ?day 30))
    ...
```

It is important to know where a granule must be searched. That is specified by the scope attribute:

- In the current indice (corresponds to the last utterance): scope="INDICE"
- In the current context (corresponds to the current unclosed context): scope="CONTEXT"
- In the current dialogue (corresponds to the entire working memory): scope="GLOBAL"

The scope can be specified for all the granules: `<conditions scope="...">` and/or for each granule: `<granule scope="...">`. Moreover, there are mechanisms of inheritance and of overload. Without inheritance, the default scope for all granules is set to INDICE that corresponds to the last utterance.
a) Manipulating the G-structures

The first use of YADEML rules is granules manipulation that allow you to define inferences on the G-structures. The possible actions on granules are: deletion, creation, disconnection, reconnection, modification, cloning. The following example is the creation and the connection of a granule [month:xx] to an existing granule [date]. This uses the get-current-month user-function:

```
<rule descr="add the current month to an incomplete date" example="the ten">
  <conditions>
    <granule concept="date" ident="?id">
      <granule concept="calendar"/>
      <nogranule concept="month"/>
    </granule>
  </conditions>
  <actions>
    <reuse-granule ident="?id">
      <create-granule concept="month: (get-current-month)"/>
    </reuse-granule>
  </actions>
</rule>
```

The second example is the replacement of a granule [tomorrow] by a [date] one:

```
<rule descr="replace tomorrow by the correct date" example="tomorrow">
  <conditions>
    <granule concept="tomorrow" ident="?id"/>
  </conditions>
  <actions>
    <replace-granule ident="?id">
      <create-granule concept="date">
        <create-granule concept="calendar: (+ (get-current-day) 1)"/>
        <create-granule concept="month: (get-current-month)"/>
      </create-granule>
    </replace-granule>
  </actions>
</rule>
```

b) Dealing with the user's utterances

YADEML provides three different methods in order to deal with the user's utterances:

- **Substring detection**: very basic but can be useful in certain cases;
- **Pattern recognition**: same formalism than YASPML (regular expressions);
- **G-structure recognition**: the more powerful method (names, offers, roles, metadata, etc.);
- **Metadata Detection**: useful method to detect particular set of words (like vulgar words)
Here are four examples of YADEML rules:

1. **Substring Detection**
   ```xml
   <rule descr="substring detection" example="Hello">
     <conditions>
       <input contains="hello"/>
     </conditions>
     <actions>
       <speak text="Hello World"/>
     </actions>
   </rule>
   
   2. **Pattern Recognition**
   ```xml
   <rule descr="pattern recognition" example="I am happy">
     <conditions>
       <input pattern="i am (very) happy/glad"/>
     </conditions>
     <actions>
       <speak text="Fantastic, me too"/>
     </actions>
   </rule>
   
   3. **Structure Recognition**
   ```xml
   <rule descr="structure recognition" example="Can I have one ticket?">
     <conditions>
       <granule concept="request">
         <granule concept="ticket">
           <granule concept="number:1"/>
         </granule>
       </granule>
     </conditions>
     <actions>
       <speak text="Please give me your destination"/>
     </actions>
   </rule>
   
   4. **Metadata Detection**
   ```xml
   <rule descr="metadata detection" example="Asshole">
     <conditions>
       <granule metadata="lang:vulgarity"/>
     </conditions>
     <actions>
       <speak text="I'm sorry but I can't accept your language"/>
       <ctrl command="stop"/>
     </actions>
   </rule>
   ```

**c) Generating the system's utterances**

YADEML offers three ways of describing the system’s answers:

- **String restitution**: very basic but perhaps the most popular way;
- **Granule rephrasing**: useful in order to discuss the user’s utterances (thanks to YAGE server);
• **Literal generation:** complete generation from a XML description (thanks to YAGE server);

This three methods can be combined in one rule. Moreover, it is possible to build the system’s utterances by parts, by the use of several rules. The final utterance will be built by concatenation of the parts.

In the case of a string restitution, some parts can be built by the use of user-functions, as in the following example, where the system inserts the result of \( \text{get-current-time} \) in its reply. Actually, all term in parenthesis is automatically evaluated:

```
<rule descr="tell the time" example="What time is it?">
  <conditions>
    <granule concept="ask-for">
      <granule concept="time"/>
    </granule>
  </conditions>
  <actions>
    <speak text="It is (get-current-time) to my watch"/>
  </actions>
</rule>
```

d) A system of nested rules

As discussed above, in YADE dialogue controller, it is possible to write nested rules in order to manage contexts of interpretation in the YADEML formalism. The following system of rules contains one initiative rule (IR) that contains one nested non-terminal rule (NR) and two nested terminal rules (NTR). For this example, please refer to the Appendix 1, shows how to manipulate the G-structures while dialoguing. The nested rule is also used in the section IV.3.2 Queueing Dialogues: Sensor Based, in the definition of "right_left_question".

**III.3 Summary**

This chapter makes us understand the basic usage of the Nao robot and goes into depth of the YADTK dialogue toolkit. The basic understanding of the usage of the robot is important as it is the main thing that we are going to operate. Then the dialogue toolkit gives the extreme facilities to program dialogues and to use the knowledge in XML formalisms is very important to know for the conduction of the thesis. And this chapter covered a thorough detailing on dialogue toolkit as well as on the robot.
Platforms: Nao and YADTK

This page is intentionally left blank for printing purpose.
IV

Integration of Nao with YADTK

The most important part of the work is to integrate Nao's SDK with YADTK software module. For this purpose understanding the software architecture of YADTK is a milestone. As seen in the previous section III.2 YADTK: A Dialogue Toolkit, the YADTK consists of several client and server modules, hence named as client-server architecture. So the target is to manipulate this client-server architecture and control the Nao's functions from NAOqi SDK as explained in the section III.1.2 Programming the Robot. This leads to control the robot by dialogue toolkit through text or speech.

In this chapter we will see the developed software architecture, different methodologies to control the robot through dialogue toolkit. Then moving to the queueing techniques and multiprocessing abilities to make the robot more and more expressive.

IV.1 Software Architecture

The Nao robot is communicated through server-client mechanism of YADTK. We have already seen YADTK software architecture. Figure 17 shown below is the same but with Nao robot, note that Nao robot is representing NAOqi SDK in the diagrams i.e. basically NAOqi is communicating through YADTK which is used to operate the Nao robot.

YADTK has two main pillars called Client and Server. This dialog toolkit has several servers and clients. NAOqi functions are called on either sides, specifically on YADE Server and Dialog Client. The general and simplified architecture that is used in the project is shown in the Figure 17.

For the sake of simplicity, we have developed Nao Client separately, which operated as a Dialog Client in YADTK, but it is specifically designed for NAOqi SDK support. The complete software architecture is shown in the Figure 18. This client has several basic
NAOqi commands written in Python called on Client side. There are also some commands on the Server side, especially on YADE Server, which are also written in Python however wrapped by CLIPS language as the core of the dialogue toolkit is written in CLIPS. This gives that the dialogue toolkit is communicating with Nao’s software NAOqi SDK on two important modules and the robot is controlled by dialog toolkit. If the robot is asked to answer any question then that question is generated by dialog toolkit and answers are defined in dialog rules as simple XML coding, as we know that the dialogue toolkit is a rule based spoken dialog system.

![Figure 17 Basic Software Architecture of Nao with YADTK](image)

As we have seen that we can call functions on the server side of dialogue toolkit, these functions are wrapped by CLIPS and returns text as output. First attempt is to utilize this mechanism to run Nao, which was successful, but there are some more functionalities that were needed to modify to make efficient and proper communication with the robot.

If the function is executed on the server side, it first runs, returns text and then outputs to the dialogue toolkit. This is the inbuilt function execution mechanism of the YADTK dialogue toolkit. In the first trial of the project we tried to implement all the functions, e.g. walking, turning, sitting, standing, getting battery information, etc. but then we realize that this function has some drawbacks. Basically, when we execute the function it get executed by suspending dialogue toolkit till it get finished.
**IV.2 Built-in Mechanism to Control the Robot**

Let us see a simple example, a simple dialogue rule written as YADEML rules in XML code. This rule has ability to generate a question “Can you sit down?” and if this question is generated it will answer “Yes, I am sitting.” and will execute the function (get-nao_sitdown) with action of sitting shown in the code below. In the “condition” tag there is an expectation of two granule concepts called “ordering” and “sit”, these are defined in next code. This was expected to run perfectly, but actual behavioral result was different.
To generate this dialogue rule, one must define the granules those are responsible for generating the above question, it is called YASPML. This is given below which shows feature declarations, e.g. “execution” and “order”. “execution” and “order” are the offers given by concept “ordering” and “sit” respectively. The concepts “ordering” and “sit” are defined to recognize the corresponding question, e.g. “can you” and “sit or sit down” respectively. Once these granules are created, one can reuse this to create several dialogue rules.

In the Python code below for the function nao_sitdown(), NAOqi SDK definitions are used. The first need is to generate the Proxies to communicate with the robot, and then one can send commands to the robot using the proxy and corresponding keyword. In this simple code we have sent a command first to go to the posture of “Sit” position with high speed (1.0 shows the speed) and then to say “Here, I am.”

```xml
<rule descr="sitting rule" example="can you sit down">
  <conditions>
    <granule concept="ordering">
      <granule concept="sit"/>
    </granule>
  </conditions>
  <actions>
    <speak text="Yes, I can sit. (get-nao_sitdown)"/>
  </actions>
</rule>

<declare-feature id="execution"/>
<declare-feature id="order"/>

<granule concept="ordering" offers="execution">
  <dependency id="A2" expected="order"/>
  <syntax pattern="can you (A2)"/>
  <syntax pattern="can (A2)"/>
</granule>

<granule concept="sit" offers="order">
  <syntax pattern="sit (down)"/>
</granule>
```
def nao_sitdown():
    global robotIP, PORT
    from naoqi import ALProxy
    postureProxy = ALProxy("ALPosture", robotIP, PORT)
    ttsProxy = ALProxy("ALTextToSpeech", robotIP, PORT)
    postureProxy.post.goToPosture("Sit", 1.0)
    ttsProxy.say("Here, I am.")
    return 
clips.RegisterPythonFunction(nao_sitdown)
clips.BuildFunction("get- nao_sitdown ", ", " , ",(python-call
nao_sitdown)"

Expected conversation:

    Human: Can you sit down?
    Robot: Yes, I can sit.
        (Robot is sitting)
    Robot: Here, I am.
        (After completion of sitting action)

Actual conversation:

    Human: Can you sit down?
    Robot: Here, I am.
        (Robot is sitting)
    Robot: Yes, I can sit.
        (After completion of sitting action, Robot is already in sitting position)

This gives rise to error in conversation and the robot miscommunicate. This example is a functions for movements, and there are some remedies to manipulate these behaviors that can be seen in the later example. Also just to mention, the robot command text-to-speech are sometime given in the functions so that the dialogues comes in the proper sequence and sometime the dialogue has to be spoken while doing an action.

One of examples is given above that on the call of the function (get-nao_sitdown), he first executes all the actions and then come out to say "Yes, I am sitting." This falsifies that example, corrected version will be seen later, but now we will see different example.

    Human: What is your battery level?
    Robot: My battery level is good it is about 56 percent.
        (Getting information about battery)
Dialogue rule is given below: YADEML

```
<rule descr="battery level rule" example="battery">
    <conditions>
        <granule concept="asking">
            <granule concept="battery"/>
        </granule>
    </conditions>
    <actions>
        <speak text="My battery level is (get-batteryLevel) percent."/>
    </actions>
</rule>
```

Granule definitions: YASPML

```
<declare-feature id="dialogue"/>
<declare-feature id="info"/>

<granule concept="asking" offers="dialogue">
    <dependency id="A1" expected="info"/>
    <syntax pattern="what is (A1)"/>
    <syntax pattern="what' s (A1)"/>
    <syntax pattern="what about (A1)"/>
</granule>

<granule concept="battery" offers="info">
    <syntax pattern="battery (level)"/>
</granule>
```

So as one can see in function batteryLevel(), it is simply defined to get battery level information from robot. If we ask “what is your battery level?” this dialogue rule will put answer awaited to get information from robot, and then it combines output speech from dialogue rule “My battery level is ______ percent.” And the black space shown is the output of batteryLevel() function. get-batteryLevel() function can output the percent available level in battery.
Integration of Nao with YADTK

And Python function (get-batteryLevel) is defined as below:

```python
def batteryLevel():
    global robotIP, PORT
    from naoqi import ALProxy
    memoryProxy = ALProxy("ALMemory", robotIP, PORT)  # Start data acquisition
    batCharge = memoryProxy.getData("Device/SubDeviceList/Battery/Charge/Sensor/Value")
    batCh = int(batCharge*100)
    if batCh < 30:  % cutoff is 30 percent
        return "low, it is about " + str(batCh)
    else:
        return "good, it is about " + str(batCh)
clips.RegisterPythonFunction(batteryLevel)
clips.BuildFunction("get-batteryLevel", "", "(python-call batteryLevel)"
```

After creating memoryProxy, it is possible to read the memory of the Nao robot from the actual Nao platform by using memoryProxy.getData("Battery Charge Sensor Value Path in ALMemory"). This value is in fraction from 0.00 to 1.00, so it is converted to integer and then in percent. For example the value is 25%, which is less than 30% so batteryLevel() function will return “low, it is about 25” or if it about 35%, which is greater than cutoff level then batteryLevel() function will return “good, it is about 35”. The returned string is combined with the string in dialog rule under speak tag that is the blank space in the sentence “My battery level is ____ percent.” So answers can be “My battery level is low, it is about 25 percent.” and “My battery level is good, it is about 35 percent.” respectively.

In this example, we have seen that the function executes first, provides information to dialogue toolkit and then outputs to the robot. Analogues to previous example, the function sitdown() executed in the same manner, just a difference is that the function batteryLevel() is providing information on the other hand the function sitdown() is defined for action of sitting. In a general point of view the function of action behavior on this of dialogue toolkit is not a good choice. One can modify these functions to work on this side as per expectation but this is not the aim of this approach. Here we want to define simple rules that can operate the robot with ease dialogue rules and robot behavior.

From these example we can conclude that the functions on the server side are best suitable for getting information from the robot. These functions are also used to get information like distance of obstacle from the robot. Let us see this example at a glance:

*Human: "What’s the distance from wall or obstacle?"

*Robot: "The distance from nearest obstacle approximately on my right is about 98 centimeters."
The dialogue rule:

```xml
<rule descr="distance rule" example="what's the distance from wall or obstacle?">
  <conditions>
    <granule concept="asking">
      <granule concept="distance"/>
    </granule>
  </conditions>
  <actions>
    <speak text="The distance from nearest obstacle approximately (get-distance) centimeters."/>
  </actions>
</rule>
```

The Python function to get distance:

```python
def distance():
    global robotIP, PORT
    from naoqi import ALProxy
    memoryProxy = ALProxy("ALMemory", robotIP, PORT)  # start data acquisition
    sonarProxy = ALProxy("ALSonar", robotIP, PORT)
    sonarProxy.subscribe("myApplication")
    # Get sonar left first echo (distance in meters to the first obstacle).
    l = memoryProxy.getData("Device/SubDeviceList/US/Left/Sensor/Value")
    # Same thing for right.
    r = memoryProxy.getData("Device/SubDeviceList/US/Right/Sensor/Value")
    #dist = int(((l+r)/2)*100)
    # Unsubscribe from sonars, this will stop sonars (at hardware level)
    sonarProxy.unsubscribe("myApplication")
    if r < l:
        return "on my right is about" + str(int(r*100))
    elif l < r:
        return "on my left is about" + str(int(l*100))

clips.RegisterPythonFunction(distance)
clips.BuildFunction("get-distance", ",", "(python-call distance)")
```

This is the way that pre-executing function works, but this wasn’t sufficient for the behavior expected. So considering the limitation of this method, there was another approach that makes the robot to speak first and then to act for the required moves. That leads to a new method for post-executing functions. In this method the robot is expected to speak first and then to make actions.
IV.3 New Mechanism to Control the Robot

In this section we will introduce post-executing functions, mostly important for action type of commands. This is a continuation of above example from the last section IV.2 Built-in Mechanism to Control the Robot, let us consider an expected conversation:

*Human: Can you sit down?*

*Robot: Yes, I can sit.*

*(Robot is sitting)*

*Robot: Here, I am.*

*(After completion of sitting action)*

The first try was on the server side, this time expose these functions on the client side. Keeping in mind that the client side of dialogue toolkit is very sensitive as it is the end-port for the communication. It is text input-output transportation module. The text is taken from the output of the dialogue toolkit and putting some code-word in the output sentence and extracting the code-word from output string to decode for commanding robot.

Example of dialogue rule for sitting action using this approach:

```xml
<rule descr="sitting rule" example="can you sit down">
  <conditions>
    <granule concept="ordering">
      <granule concept="sit"/>
    </granule>
  </conditions>
  <actions>
    <speak text="Yeah, I can sit. [nao:sitdown]"/>
  </actions>
</rule>
```

Note: The concepts “ordering” and “sit” are defined as granules in the previous example are same and reused in this dialogue rule. This is an important feature of the dialogue toolkit that we don’t need to redefine the granules. Once it is defined and created, one can reuse it in infinite dialogue rules.

Just to mention that the code-word is defined in [] followed by “nao:”, e.g. [nao:sitdown]. Here the code-word is “sitdown” and it has a corresponding Python function as earlier. But this Python function is little easier, because there is no need to wrap up this Python function in CLIPS language, as it is called on Python level client of the dialogue toolkit.
The extraction of this code-word is carried out at the output module of the client. As we have seen in the section of YADTK that there are several input and output modules which are connected to client of the dialogue toolkit. The new output module called Nao Output module is specifically designed to output text on Nao’s text-to-speech (TTS) module and also to manage the extraction and decoding of the code-words by using Regular Expression library of the Python. After this step these codes are defined to specific actions. The logic used for this technique is very simple, it says that for example if code-word “sitdown” is found run the function related to this code-word.

The ‘output’ function of the YADTK, written for Nao Output is as given below:

```python
def _output_(text):
    # For example the YADE action is
    # <speak text="Ok man, I can do that [nao:walk]/">
    # Suppressing the command tags in the text.
    # E.g. "Ok man, I can do that"
    tosay = re.sub(r"\[\^[^\]]*\]\", "", text)

    # Sending the text to the Nao's TTS (cf. nao_commands.py)
    talk(tosay)

    # Searching for a Nao command tag in text.
    # E.g. “walk” from above string.
    result = re.search("\[nao:(.*?)\]", text)

    if result:
        # Executing the action if there is one
        command = result.group(1)
        if command == "standup": standup() #1
        elif command == "sitdown": sitdown() #2
        elif command == "turnLeft": turnLeft() #3
        
Once, the output of dialogue toolkit is generated, it comes out as text to the _output_(text), and then this text is converted into two parts. The first part is “tosay” variable, this variable is extracted from output “text” by using regular expression re.sub() that suppress out any of the commands like “[nao:walk]”. This variable string “tosay” is then passed as input to the talk() function. This talk() function is to convert the text into speech, but using Nao’s text-to-speech (TTS) module, so that in actual Nao will speak out. The talk() function is a Python code defined in the naoqi module in the nao_commands.py, where all other commands are also defined such as standup(), sitdown(), etc.

The second part is to search the command coded in the output string. Next variable called “result” will search the code-word encoded in the output string, by using again regular expression re.search() function from “text”. It will search to the word written in the
Integration of Nao with YADTK

brackets [] after “nao:”, format is [nao:____here_is_code-word____]. For example, in the expression [nao:sitdown], “sitdown” is code-word.

If any of the code word is found, the corresponding Nao’s functions are assigned to each code. The re.search() will search such command tags in the output text, and if there is one found, it is converted into command and corresponding function is executed. Note that if there is not a single command found in the output string then there will be no action an Nao robot will speak the variable “tosay” using talk() function.

In our example we have an output text:

   <speak text="Yeah, I can sitting. [nao:sitdown]"/>

Here, the variable
tosay = "Yeah, I can sitting."
command = “sitdown”.

This gives rise to the execution of function “sitdown()”. This function is defined in naoqi module in the nao_commands.py.

```
def sitdown():
    global postureProxy
    if str(postureProxy.getPostureFamily()) == "Sitting":
        talk("but, I am already in sitting position")
        return ""
    cprint("==> Nao is sitting down...", "green")
    postureProxy.post.goToPosture("Sit", 1.0)
    talk("I am sitting.")
```

As shown in the function sitdown(), firstly it checks that if the robot is already in sitting position by the command postureProxy.getPostureFamily() of postureProxy proxy. This returns the current posture of the robot, if robot is already in sitting position he can anticipate that he is already in sitting position. If he is not in sitting posture, then it is asked to go into “Sit” posture by using postureProxy.post.goToPosture("Sit", 1.0).

Now, coming back to our example, if the robot is asked to “can you sit down?” then he can answer in two manners. First, if he is in sitting position then he can answer that he is already sitting position or if not, then he will sit down.
Conversation can be as given below:

First: if it already in sitting posture.

*Human: Can you sit down?*

*Robot: Yes, I can sit.*

*(Robot is checking his posture, making sure him not in sitting position)*

*Robot: but, I am already in sitting position.*

*(as he in sitting action)*

Second: if it is not in sitting posture.

*Human: Can you sit down?*

*Robot: Yes, I can sit.*

*(Robot is checking his posture, making sure him not in sitting position)*

*Robot: I am sitting.*

*(while sitting down)*

So, this example shows lot of improvements in the sequence of the conversation without any miscommunication. One can also notice that the facility given in NAOqi SDK functions to know about current posture of the robot, which can be used to avoid miscommunication, for example, if we ask the robot to sit down, if he doesn’t know he is already sitting position, then also he will say “I am sitting”, which is wrong on conversation point of view. The understanding of this ability and availability of such functions is very important to make natural verbal communication.

**IV.4 Methods to Call Functions on Server/Client Side**

Now, it is well known that we can call the function on the server side and/or the client side. Let us explore now, how we can call the functions with their behaviors.

Basically, there is chance to classify these two functions may be called as pre-executing and post-executing function called on the server and the client side respectively.

There is the possibility to call or operate the robot on either server side or client side that lead to consider as a taxonomy of functions for classifications as they have different behaviors.

Functions called on Server Side (YADE server) are Pre-executing.

Functions called on Client Side (Dialog Client) are Post-executing.
IV.4.1 Pre-executing Functions on Server Side

This type of functions is defined on the server side and especially on YADE (Yet Another Dialogue Engine) server of dialogue toolkit. Python functions are defined at once, and reused as per requirement. The Python functions are wrapped up in CLIPS language, because primarily speaking that the functions called on the server side are written in CLIPS language. But thanks to the wrapping feature of CLIPS, that we can wrap up a Python function and execute the NAOqi Python functions.

To understand more, have a look on the example below, shows how to wrap the Python function in CLIPS. The NAOqi Python functions are defined in the application.py file and they are registered in CLIPS in application.clp file.

As one can see in application.py, we have defined two functions gethuman() and batteryLevel(). These functions are followed by clips.RegisterPythonFunction() command which is used to register the corresponding Python function and then clips.BuildFunction() to build it. These are the common commands that are required to follow for any function.

Once the Python functions are defined and build in the application.py file then application.clp file defines that function using (deffunction MAIN::get-functionName ()) command. Now this function can be called by client of dialogue toolkit. For example if the YADEML dialogue rule is written as:

In the yaderule.xml

```
<rule descr="human presence" example="what about human presence">
   <conditions>
      <granule concept="asking">
         <granule concept="human presence"/>
      </granule>
   </conditions>
   <actions>
      <speak text="There (get- human)."/>
   </actions>
</rule>
```

In the application.clp

```
(deffunction MAIN::get-human())
   ;; This function defined in the file application.py as gethuman()

(deffunction MAIN::get-batteryLevel())
   ;; This function is defined in the file application.py as batteryLevel()
```
In the application.py

```python
% Here are the functions that are called on demand
def gethuman():
    numHuman = trackHumans()
    if numHuman == 1:
        return "is one human"
    elif:
        return "are humans"
clips.RegisterPythonFunction(gethuman)
clips.BuildFunction("get-human", "", "(python-call gethuman)")

def batteryLevel():
    global robotIP, PORT
    from naoqi import ALProxy
    memoryProxy = ALProxy("ALMemory", robotIP, PORT)
    batCharge = memoryProxy.getData("Device/SubDeviceList/Battery/Charge/Sensor/Value")
    batCh = int(batCharge*100)
    if batCh < 30:
        return "low, it is about " + str(batCh)
    else:
        return "good, it is about " + str(batCh)
clips.RegisterPythonFunction(batteryLevel)
clips.BuildFunction("get-batteryLevel", "", "(python-call batteryLevel)")
```

If the question is generated “What about human presence?” answer can be generated as “There _____”, so this blank space can be filled by using the function (get-human), which actually a CLIPS function but will run the Python function that was wrapped and built. When the function is found in the tag “speak” i.e. text="There (get- human)." then the output goes on hold till the function is being executed. After execution of the function, the return string goes to the blank location, for example the output string can become “There are humans.” or “There is human.” depending on the output of sub-function trackHumans(). Note that this function does not exist, it is an example that is it is possible to call other function inside a main function.

In the same manner for the second function batteryLevel(), there can be two outputs such as “low, it is about 25” is battery level is less than 30 percent or “good, it is about 35 percent” if it is greater than 30 percent. This is an active example implemented on robot and gives good result, the conversation is shown:
Integration of Nao with YADTK

If battery level is greater than 30 percent:

*Human:* What is your battery level? Or What about your battery?

*Robot:* My battery level is good it is about 56 percent.

If battery level is less than 30 percent:

*Human:* What is your battery level? Or What about your battery?

*Robot:* My battery level is low it is about 26 percent.

**IV.2.2 Post-executing Functions on Client Side**

It is already explained in the section **IV.2 Built-in Mechanism to Control the Robot** that because of miscommunication occurred in the conversation because of pre-executing behavior if the need is to get actions from the robot like sitting, standing, walking, etc. To fulfill this need we have modified the output plug-in designed for the Nao robot. The robot is operated on end output text level of the dialogue toolkit, this is the fundamental of the post-executing functions. This function is named as post-executing because dialogue toolkit is able to give speaking action first and then the function is executed.

The Python functions are defined in the modules, named as naoqi, by using NAOqi SDK functions. There are several Python scripts such as naoqi.py, nao_commands.py, nao_loops.py and nao_findWall.py. Each of these scripts has specific type of functions, for example, naoqi.py script defines all the required proxies globally such as TTS (text-to-speech proxy), motionProxy (motion proxy), postureProxy (Robot posture proxy), memoryProxy, sonarProxy, cameraProxy, etc. The robot IP, PORT and Volume control are extracted that are set in the configuration.xml file of the dialogue toolkit, which is explained in detail in the section of dialogue toolkit.

The naoqi.py file generates all proxy variables that can be access in all other functions globally. This is made possible with the help of exesfile() function of Python. The main interface to all these function is in the nao_output.py file where one can find the footage of all these files, such as:
As one can notice that these files are globally accessed in Nao’s output plug-in. This shows the code of the previous example, but it completes with execfile(“all Nao’s files from module naoqi”). If any of the command found in the “result” variable group then the functions like “standup()”, “turnLeft()”, etc. are called from naoqi module files. The functions for various movements or actions such as sitting, standing, walking, turning, etc. are defined in nao_command.py file.
The Python script nao_loops.py has some loops that are running all the time parallel to other tasks, which checks continuously the state of the battery level, stiffness, etc. This is to ensure that, for example, the battery level should not drop below 20 percent, and if the battery level goes below this level, then the robot is able to tell us that the battery level is very low we must charge it. These loops are explained in the section IV.4 Multiprocessing Functionality.

The next Python script is nao_findWall.py, this file has various functions that starts camera of Nao, reads QR code from the images taken, navigate to the found wall, can find a particular wall or QR code in general and navigate to it. This is explained in detail in the next chapter Advance Task Specification and Implementation in section V.1 Description.

In this section, the main focus given on Nao commands. These commands are defined by using NAOqi SDK. Some of the examples are given in above code.

```python
# Function to make the robot standup
def standup():
    global postureProxy
    if str(postureProxy.getPostureFamily()) == "Standing":
        talk("but, I am already in standing position")
    return 
    cprint("==> Nao is standing up...", "green")
    postureProxy.post.goToPosture("StandInit", 0.7)
talk("I am standing.")

# Function to make the robot sit-down
def sitdown():
    global postureProxy
    if str(postureProxy.getPostureFamily()) == "Sitting":
        talk("but, I am already in sitting position")
    return 
    cprint("==> Nao is sitting down...", "green")
    postureProxy.post.goToPosture("Sit", 0.7)
talk("I am sitting.")

# Function to make the robot turn left
def turnLeft():
    global motionProxy, postureProxy
    cprint("==> Nao is turning left...", "green")
    postureProxy.goToPosture("StandInit", 1.0)
    motionProxy.post.setWalkTargetVelocity(0.0, 0.0, 1.0, 1.0)
talk("I am turning left.")
time.sleep(1.5)
motionProxy.setWalkTargetVelocity(0.0, 0.0, 0.0, 0.0)

# Write functions likewise to make robot act/move like you want: here is a pseudo function
def pseudo_functions():
    global "All required proxies..."
    check the robot conditions...
    write command to make act/move robot...
    stop the motion if any...
```
These are very simple functions written to make robot move and walk. Alongside to it, there is another beautiful feature of dialogue toolkit that is to put dialoguers in queue, which we will see in the section IV.3 Queueing Dialogues. It is also possible to call both type of functions at the same time, next topic gives an example how one can utilize this methodologies to call both the functions in the one question.

**IV.2.3 Concurrent Execution of the Functions on both sides**

This section in generalized for utilizing both the functions in the single dialogue. This is possible to do because of granularity of the natural language generation understanding of the dialogue rules. For example of the robot asked to know its battery level and at the same time asked to move right/left, the robot must respond to both questions/orders and also anticipate to answers. From above examples, see the following conversation.

```
Human: What about your battery level? And turn left?
   (Notice these are two different questions/orders)
Robot: My battery level is good, it is about 56 percent. Yes, I can turn left.
   (The robot getting information for battery level and also following order)
Robot: I am turning left.
   (While turning)
```

Or second example, where one can ask robot about distance from obstacle and robot must understand that he can move toward that direction or not.

```
Human: What is distance from obstacle, can you move toward it?
Robot: The distance from obstacle is 1.5m, I can move toward it,
do you want me to move?
Human: Yes, please.
Robot: Okay, I am moving, I will stop near the obstacle.
```

So as we can see in these examples, that the robot is able to use its sensor data and also interact with the user for the anticipating next action by using that data. This is so fascinating to notice that the robot is capable of anticipation at the ground level programming. This gives an ability to program sophisticated features for the robot and utilize all its sensors to make artificial intelligent assistantship.
IV.3 Queueing Dialogues

This is another important feature of the dialogue toolkit on real time that we put dialogue rules in the queue to speak out. There are two possibilities to put the dialogues in a queue: time based and sensor based.

IV.3.1 Queueing Dialogues: Time Based

In this case, the dialogues can be putted in the queue to speak after particular amount of time. For example if the robot is not being communicated for a while say 60sec, then he can speak something to show his existence. E.g. suppose that the robot is idle for 60sec, so he can ask human:

```python
def naoInitiative():
    global queue # All inputs must go through the queue
    if queue=empty:
        time.sleep(60)
        queue.put("nao_initiative_on_silance")
```

This function checks if the queue is empty. If it is true that the queue is empty for 60 sec the dialogue rule "nao_initiative_on_silance" is pushed in the queue by using queue.put(“rule”). The dialogue rule "nao_initiative_on_silance" is given in the Appendix 4. The produced conversation can be as shown below:

Robot: Hey, is there anybody? Speak with me.
Human: Oh, what’s up? I was working.
   (The user can respond to the robot if there is any)
Robot: Okay, I thought you forgot me.
Human: No, that’s not a case my friend.
Robot: Okay, continue your work, thanks for being with me.
Human: See you soon.

Or if there is nobody, robot can speak out, sit down, and loose his stiffness to save battery.

Robot: Hey, is there anybody? Speak with me.
Human: (Not responding, suppose nobody is present there)
   (If the robot is not in sitting position)
Robot: Oh, I think nobody is there. I must sit down, my motor gets harder.
The robot can turn on stiffness if somebody call him and also will ask user that nobody was there, for example:

- **Human**: Hey robot, how are you?
- **Robot**: I am fine. I was sleeping as there was nobody from long time.
- **Human**: That’s great.
- **Robot**: Thank you.

### IV.3.2 Queueing Dialogues: Sensor Based

This is another feature that has a lot of importance for utilizing sensory feedback to put dialogues in the queue. The simple example, if we touch head tactile sensor of the robot, the robot must be able to respond something.

- **Human**: (touches head of the robot).
- **Robot**: Hey, that’s my mind. Don’t touch my head! (The robot is looking upward)

Or second simple example can be hitting the bumpers of the leg of the robot.

- **Human**: (hits the bumpers of the robot).
- **Robot**: Ouch, you hurt me. (The robot looking at the leg which you hit)
- **Human**: Oh, I am sorry.
- **Robot**: That really hurts me.

These two examples are implemented by using the multiprocessing feature of the dialogue toolkit which is explained in the next section IV.4 Multiprocessing Functionality, where Nao’s looping system is used (defined in Python script nao_loops.py).

The looping functions defined to read data from corresponding sensors continuously, and on occurrence of particular condition the dialogue rules can be triggered or can be queued to speak by the robot. Let us see the implementation of first example of touching head. The loops for above example, the Python script function can be defined as:
The Python function defines the behavior of the robot, it continuously read the tactile sensor data in an infinite “while” loop and if the any of sensor data goes high i.e. if somebody touch robot’s head, the robot says “Hey!” and puts the “touch_head_rule” dialogue rule in the queue.

In yaderules.xml, the dialogue rule can be written as:

```xml
<rule descr="touching head rule" example="if somebody touches head">
  <conditions>
    <input contains="touch_head_rule"/>
  </conditions>
  <actions>
    <speak text="There is my mind. Don’t touch my head!"/>
  </actions>
</rule>
```

As one can see in this YADE rule, the tag “input” has command contains=“rule”, here “touch_head_rule” which is pushed into the “queue” by using queue.put("rule") command in Python script. In the same manner one can implement second example as well. Now, let us have a look on another example of “can you go forward?”, in this case user is asking robot to go forward. Robot is expected to go forward and there is possibility that he can found an obstacle and the robot can stuck somewhere.
This is a little complicated to implement, in this example it is expected for robot to go forward, find obstacle (using sonar sensors), if the robot found obstacle then ask user that there is obstacle should he go on his left or right.

At the first let us have a look on the possible conversation, given below:

**Human:** Can you go forward? (Or just “Go forward.”)

**Robot:** Okay, let’s go. I will stop if there is an obstacle. (Robot is walking forward)

**Robot:** Oh, there is obstacle in front of me, should I go on my left or right?

(The robot was checking obstacle with sonar sensor, and this dialogue kept in queue on occurrence of obstacle)

**Human:** What is distance from obstacle?

(Verbal interrupt: asking something else not related)

**Robot:** The obstacle is 45cm away. Please answer me, left or right?

(Answers a question but remembers previous state, thanks to the nested dialogue rules and terminal and nonterminal definitions: shown in the corresponding dialogue rule below)

**Human:** You can go to the left.

**Robot:** Okay, let’s go to the left.

Now, let us split the dialogues into different stages, first we ask robot to move forward. The dialogue rules can be written as:

```xml
<rule descr="move forward rule" example="can you move forward">
  <conditions>
    <granule concept="ordering">
      <granule concept="forward"/>
    </granule>
  </conditions>
  <actions>
    <speak text="okay, let's go.[nao:walkForward]"/>
  </actions>
</rule>
```

Note that the granules for “ordering” and “forward” are defined in the same manner as it is written in section IV.2 Built-in Mechanism to Control the Robot. From this YADE dialogue rule once the robot is asked to move forward, the robot says “okay, let’s go.” and execute the function walkForward() on the occurrence of the code-word “walkForward” as explained in the section Post-executing functions.
The function walkForward() is a Python script defined as:

```python
def walkForward():
    global motionProxy, postureProxy, memoryProxy, sonarProxy, queue
    cprint("==> Nao is walking forward...", "green")
    postureProxy.goToPosture("StandInit", 1.0)
    motionProxy.post.setWalkTargetVelocity(1.0, 0.0, 0.0, 1.0)
    talk("I will stop if there is obstacle.")
    while True:
        time.sleep(0.1)
        # Subscribe to sonars, this will launch sonars
        # (at hardware level)
        # and start data acquisition.
        sonarProxy.subscribe("myApplication")
        # Now you can retrieve sonar data from ALMemory.
        # Get sonar left and right first echo
        # (distance in meters to the first obstacle).
        l = memoryProxy.getData("Device/SubDeviceList/US/Left/Sensor/Value")
        r = memoryProxy.getData("Device/SubDeviceList/US/Right/Sensor/Value")
        # Unsubscribe from sonars, this will stop sonars
        # (at hardware level)
        sonarProxy.unsubscribe("myApplication")
        minDist = 0.5
        if l < minDist or r < minDist:
            motionProxy.setWalkTargetVelocity(0.0, 0.0, 0.0, 0.0)
            queue.put("right_left_question")
            break
        time.sleep(0.2)
    motionProxy.setWalkTargetVelocity(0.0, 0.0, 0.0, 0.0)
```

As shown in the function, the robot moves forward and checks for obstacle by reading sonar sensor data continuously in infinite “while” loop. If there is an obstacle found in the range of minDist, i.e. minimum distance from the obstacle, then the dialogue rule "right_left_question" is pushed into the queue by using queue.put() command.
"right_left_question" dialogue rule can be defined as:

```xml
<!-- Example of a system of rules (navigation from Nao) -->
<rule descr="right-left question">
  <conditions>
    <input contains="right_left_question"/>
  </conditions>
  <actions>
    <speak text="Oh, there's a obstacle in front of me."
    <speak text="Should I go on my right or on my left? [timeout:5]"/>
  </actions>
</rule>

<!-- Right direction: Terminal rule -->
<rule terminal="TRUE" descr="right choice" example="to your right">
  <conditions>
    <granule concept="right"/>
  </conditions>
  <actions>
    <speak text="OK, let's go to the right. [nao:turnRightWalk]"/>
  </actions>
</rule>

<!-- Left direction: Terminal rule -->
<rule terminal="TRUE" descr="left choice" example="to your left">
  <conditions>
    <granule concept="left"/>
  </conditions>
  <actions>
    <speak text="OK, let's go to the left. [nao:turnLeftWalk]"/>
  </actions>
</rule>

<!-- Non-terminal rule -->
<rule descr="no direction" example="hello">
  <conditions>
    <nogranule offer="direction"/>
  </conditions>
  <actions>
    <speak text="Please answer me: right or left? [timeout:5]"/>
  </actions>
</rule>
</rule>
```

In this dialogue rule, there is use of nested rules, and also terminal-nonterminal rules. When the robot found an obstacle this dialogue rule is triggered and when this rule is pushed into queue there are two choices to go either in left or right. The choices can be fulfilled by the terminal rules as defined in the dialogue rules. For example, user can say left or right depending on navigation condition. For instance if user says left, then robot should turn left and move towards new direction and same for the right direction choice. The dialogue must be terminated here, this is possible using terminal dialogue rule. The corresponding functions turnRightWalk() and turnLeftWalk() are in the Python function nao_commands.py.
If the user doesn’t answer the question, the robot waits for 5 sec defined by “[timeout:5]” and after 5 sec asks user “Please answer me: right or left?”. Another case is that if the user asks something else that is not related to current topic, for example “What is distance of obstacle?” the robot answers this new question and also remembers the previous state. This is possible because of Non-terminal dialogue rule.

**IV.4 Multiprocessing Functionality**

Sometime there is a need to process some functions in parallel with processing of the dialogue toolkit. For example, checking battery level continuously and if it goes down to particular level the robot must intimate user to charge it. This kind of functions can be processed in parallel without affecting the current process of the dialogue toolkit. This ability makes robot more interactive, responsive and self-conscious.

This feature is adapted from the Python library named as “Multiprocessing Package”. This package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using sub-processes instead of threads. Due to this, the multiprocessing module allows the programmer to fully power multiple processors on a given machine.

Some of the recognized tasks are checking battery level, heat sensor values, stiffness etc. The Python scripts are written in nao_loops.py file, where different functions checks different values continuously in steps defined. Here is a one example to show the implementation of the battery level checker, the corresponding conversation could be like:

If robot battery goes below 20% and user is present.

*Robot: Hey there, my battery level is very low, please charge me.*

*Human: Okay, I am putting a charger.*

*Robot: Thank you. I was about to die.*

*Human: You are welcome.*

If robot battery goes below 20% and user is absent.

*Robot: Hey there, my battery level is very low, please charge me.*

*(As no user responded the robot keep saying the following dialogue for every 5 sec.)*

*Robot: Please, charge my battery, otherwise I will die.*

*(The robot will keep saying this and will switch off after discharge of battery)*
The Python function is written in nao_loops.py:

```python
def batteryLevelChecker():
    global robotIP, PORT, queue
    from naoqi import ALProxy
    memoryProxy = ALProxy("ALMemory", robotIP, PORT)
    while True:
        time.sleep(1)
        batCharge = memoryProxy.getData("Device/SubDeviceList /Battery/Charge/Sensor/Value")
        batCh = int(batCharge*100)
        if batCh < 20:
            queue.put("intimation_for_bat_charging")
            break
```

Multiprocessing.batteryLevelChecker()

The Python script shown here explains that it is continuously reading the battery charge level in a “while” loop for every second (1 sec) and if the level drops below 20 percent the queuing technique is used to start dialogue rule as detailed in section IV.3 Queueing Dialogues. The yade-rules defined under “input” tag as contains = “intimation_for_bat_charging” which is the dialogue rule to intimate the user to make the robot charge battery.

YADE dialogue rule written in yaderule.xml:
This kind of feature can be implemented to make robot self-conscious and demand emergency actions from the user. Also important to note that the robot can take self-decision on the basis of conditions provided.

**IV.5 Summary**

In this chapter, the main part of the work is detailed. The approaches are unique and all the methodologies are implemented. We have seen that the Nao robot is controlled by a dialogue toolkit, YADTK. Several methods and features are explored. Most important to note about pre-executing and post-executing functions called on the server and the client side of the dialogue toolkit, respectively. Then most important features of queueing the dialogues on the basis of time or sensory feedback. Also to make the robot self-aware the multiprocessing functionality gave the robot an important ability.
Navigating Nao with Verbal Aid:
Task Implementation

The verbal dialogue toolkit is now well interfaced with the humanoid robot Nao. In this chapter, the Nao robot is asked to do some advance task. The robot can follow orders from the human user and is able to communicate in the natural language. Now, we can ask the robot to read Quick Response (QR) codes using vision, and utilize this ability to navigate the robot. Different QR codes are assigned to walls and Nao is able to recognize wall with the help of these codes. Then the user can ask to find the particular wall, and to follow the same.

In this chapter, we will see the implementation of the particular task that will make the robot to explore the environment by using the vision. The corresponding sections will focus on how to generate QR code, how to read them, and then search them in the environment. On the occurrence of the code asking the robot to follow any of them. This makes the robot to navigate in the known environment with the verbal guidance of human user companion.

V.1 Description

The idea is to guide the robot verbally, asking robot to find walls or objects with the help of the QR code and then follow that object. Presuming that the robot knows QR codes available in the environment. The QR codes are assigned to walls and objects. The robot is able to read these codes from at least 0.5 meter to 3 meters, it is dependent on the camera resolution.
V.2 Generating the QR Code

The generation of the QR code is achieved with the help of the qrcode Python library dedicated to QR Code image generator. A Quick Response (QR) code is a two-dimensional pictographic code used for its fast readability and comparatively large storage capacity. The code consists of black modules arranged in a square pattern on a white background. The information encoded can be made up of any kind of data (e.g., binary, alphanumeric, or Kanji symbols). This module uses image libraries, Python Imaging Library (PIL) by default, to generate QR Codes.

Simple Usage:

The code can be generated either by using the command line, use the installed qr script:

```plaintext
qr "Some text (coding data)" > test.png
```

Or in Python, use the make shortcut function:

```python
import qrcode
img = qrcode.make('Some data here (coding data)')
```

Advanced Usage:

For more control, one can use the QRCODE class in Python. For example, here is a code to generate QR code with “FirstWall” encoded on it and saved with the same name in bmp image format as shown in the code below.

The version parameter is an integer from 1 to 40 that controls the size of the QR Code (the smallest, version 1, is a 21x21 matrix). Set to None and use the fit parameter when making the code to determine this automatically.

```python
import qrcode
import Image

qr = qrcode.QRCode(
    version=1,
    error_correction=qrcode.constants.ERROR_CORRECT_L,
    box_size=10,
    border=4,
)
qr.add_data('FirstWall')
qr.make(fit=True)

img = qr.make_image()
img.save('/home/crb/test/FirstWall.bmp', 'bmp')
```
The `error_correction` parameter controls the error correction used for the QR Code. The following four constants are made available on the `qrcode` package:

```
ERROR_CORRECT_L
    About 7% or less errors can be corrected.
ERROR_CORRECT_M (default)
    About 15% or less errors can be corrected.
ERROR_CORRECT_Q
    About 25% or less errors can be corrected.
ERROR_CORRECT_H.
    About 30% or less errors can be corrected.
```

The `box_size` parameter controls how many pixels are present in each “box” of the QR code. The `border` parameter controls how many boxes thick the border should be (the default is 4, which is the minimum according to the specs).

And finally, `qr.make_image()` and `img.save()` commands are used to make and save image respectively.

The generated QR codes for data “First Wall” and “Second Wall” are as given below in Figure 19:

![Generated QR code for data “First Wall” and “Second Wall”](image-url)
Several codes are generated and printed on the A4 size papers. These printed codes are attached to the corresponding walls and the robot is trained to read the codes using *ZBar* bar code reader library of Python, detailed in the next section.

### V.3 Reading the QR Code

The QR code can be read with help of *ZBar* package library of Python. *ZBar* Bar Code Reader is an open source software suite for reading bar codes from various sources, such as video streams, image files and raw intensity sensors. It supports EAN-13/UPC-A, UPC-E, EAN-8, Code 128, Code 39, Interleaved 2 of 5 and QR Code. These are the Python bindings for the library.

In case of the work, the generated QR code is read using *ZBar* package with the help of image files streamed from the Nao’s front camera. The Nao robot was first trained to read different codes from the front camera. Images are fetched with the help of *naoqi* library by using camProxy (camera proxy), which is responsible to communicate with robot for getting images from the robot camera.

The function below `getImage()` is given below that get image from the robot camera and save the image in local directory on computer with name “camImage.png” and this image is then used to scan for reading QR code.

```python
def getImage():
    global camProxy
    resoltn = 2  # VGA
    colrSpc = 11  # RGB
    videoClient = camProxy.subscribe("python_client", resoltn, colrSpc, 5)
    t0 = time.time()
    # Get a camera image.
    # image[6] contains the image data passed as an array of ASCII chars.
    naoImage = camProxy.getImageRemote(videoClient)
    t1 = time.time()
    # Time the image transfer.
    #print "acquisition delay ", t1 - t0
    camProxy.unsubscribe(videoClient)
    # Get the image size and pixel array.
    imageWidth = naoImage[0]
    imageHeight = naoImage[1]
    array = naoImage[6]
    # Create a PIL Image from our pixel array.
    im = Image.fromstring("RGB", (imageWidth, imageHeight), array)
    # Save the image.
    im.save("camImage.png", "PNG")```
Once the image is captured, it is ready to fetch for the scanning, to read the QR code if available in the image. Note that, the libraries “Image” or “zbar” are imported in the main file, naoqi.py. These two libraries are used in the following function scanqr().

The function scanqr() scans the image fetched by the function getImage() from the robot camera. The same image name "camImage.png" is called here from the local directory. This function returns the variable “sym”, which is a corresponding symbol read from the image.

As one can see that the function scanqr() scans the image from the Nao robot camera and read the code available in the image if any. The robot is trained to read the codes from different images with different angles and positions in the image.

```python
def scanqr():
    # Scanning image for reading QR code: create a reader
    scanner = zbar.ImageScanner()

    # configure the reader
    scanner.parse_config('enable')

    # obtain image data
    pil = Image.open("camImage.png").convert('L')
    width, height = pil.size
    raw = pil.tostring()

    # wrap image data
    image = zbar.Image(width, height, 'Y800', raw)

    # scan the image for barcodes
    scanner.scan(image)

    # extract results
    for symbol in image:
        # Return the results: Symbol (sym) the code in the image
        sym = symbol.data
        #print 'decoded', symbol.type, 'symbol', '"%s"' % symbol.data
        return sym

    # clean up
    del(image)
```

The images shown in Figure 20 are captured by using the robot’s front camera. The robot is trained to recognize and read the QR code from these images. The robot is made to move around and ask to find particular code or wall in the environment.
V.4 Navigation by Reading QR Code

The robot is now able to read the QR code and can output/return the coded data on the image. The basic idea implemented for navigation is to find the QR code in the images captured by the camera of the robot an on the occurrence of a particular code, the set of actions is performed. The actions are set to follow the object (the QR code) found in the image.

V.4.1 Searching QR Code while Rotating the Head

For example, the robot is rotating his head and scanning the images, if the QR code is found in the image, the angle of the head is recorded. This recorded angle is used to turn whole body of the robot towards the code found. Once the robot is turned to the code, then it can
be asked to move toward the recognized code.

```python
def headDir(direction):
    if direction == "right":
        return 1.6, 0.0
    elif direction == "left":
        return 0.0, 1.6
    elif direction == "front":
        return 1.6, 1.6
    elif direction == "straight":
        return 0.4, 0.4
```

As shown in the above code, the function `headDir()` is defined to get the direction of head to rotate with respect to the variable ‘xx’. The variable ‘xx’ is varied from the limits ‘right’ to ‘left’. The variable ‘right’ and ‘left’ shows the limit of the head on either side. The definition of these limits allows the robot to reduce the search area. Then, the user can ask “find the wall in the right direction”, so the robot will search only in the defined area for the right direction, as shown in Figure 21:

![Figure 21 Head Directions for searching the object](image)

The variable direction is dedicated to put “right”, “left”, “front” and “straight”, and the corresponding actions are as shown in Figure 21. The values returned by the function are in radians and correspond to the yaw angle of rotation of the head in right and left direction.

The Python function `findReqWall()`, Find Required Wall, is defined to find the wall while rotating the head, so the robot is expected to rotate his head and search for the wall (a QR code).
around him. The input of the is ‘findWall’ and ‘direction’ i.e. findReqWall(findWall, direction). There are two outputs of the function: firstly the function puts a dialogue rule in the queue and secondly it return the variable ‘xx’ which is the yaw angle of the head. We already have seen, how to use the variable ‘direction’, so that the robot will search in that area only. The variable ‘findWall’ signifies which wall is asked to search. For example, one can call the function like:

```python
headAngle = findReqWall("FirstWall", "right")
```

This function will find “FirstWall” in “right” direction. The QR code for “FirstWall” is shown in Figure 19 and area covered for the “right” direction is given in Figure 21. The execution will put (“move_to_wall”), is given in V.4.2 Moving Towards

```python
def findReqWall(findWall, direction):
    global camProxy, motionProxy, postureProxy, memoryProxy, sonarProxy
    
z = headDir(direction)
    right = z[0]
    left = z[1]
    
    for xx in frange(-right, left, 0.2):
        # The robot is rotating head Yaw angle with variable xx
        motionProxy.setAngles("HeadYaw", xx, 0.9)

        # Get image from nao camera per step in the variable xx
        getImage()

        # Scan each image for reading QR code
        sym = scanqr()

        # Check whether the required wall is found
        if sym == findWall:
            # Move head to initial position
            motionProxy.setAngles("HeadYaw", 0.0, 0.9)

            talk("I found the required wall.")

            # Put the dialogue rule in the queue
            queue.put("move_to_wall")

            return xx

    # If robot found nothing then this function returns following
    if (xx-left < 0.01) and (sym == None):
        talk("I didn't found any wall, ask me to turn and then to find again")

        #return "Nothing", xx
        return "Nothing", xx
```

the found QR Code, dialogue rule in the queue for further conversation and the function will return “xx” as a new variable “headAngle”.

Now let us see how the function works. In the function findReqWall(findWall, direction) function uses all global proxies to establish communication with the robot.
Then it uses the function `headDir()` function to get the search direction of the robot as given above in Figure 21. Then ‘right’ and ‘left’ variables are generated for the direction and the variable ‘xx’ is varied in between these limits by using function `frange()` defined in the Appendix 2., which allow fractional increment of float values. Each iteration of ‘xx’ is corresponds to the yaw angle of the rotation of the head.

For each values of ‘xx’ in the for loop, the robot captures an image by using `getImage()` function and scan this image for the QR code by using `scanqr()` explain in the section V.3 Reading the QR Code. , that return variable 'sym' i.e. symbol in the QR code.

This symbol is then compared with the input variable ‘findWall’ if it is found then the dialogue rule "move_to_wall" is pushed into the queue and the Yaw angle of the head is recorded by returning the ‘xx’ variable. And if the robot is not able to find any wall he returns the current and says "I didn’t found any wall, ask me to turn and then to find again.”

The main dialogue rule to ask to find the wall in given below in code below:

```xml
<rule descr="find first wall rule" example="can you find/search first wall">
  <conditions>
    <granule concept="ordering">
      <granule concept="findfirstwall"/>
    </granule>
  </conditions>
  <actions>
    <speak text="Yeah sure, I will search the first wall around me. [nao:findfirstwall]"/>
  </actions>
</rule>
```

Note that, the granules "ordering" and "findfirstwall" are defined in the Appendix 3. Also to mention that the granule "ordering" is reused from the example in section IV.2 Built-in Mechanism to Control the Robot.

The user can ask the robot to find the wall, the conversation can be like:

_Human: Can you find the first wall?_

_(Or Find first wall)_

_Robot: Yeah sure, I will search the first wall around me._

_(The robot starts rotating head and searches for first wall)_

The robot will search first wall in the environment because of the code-word that is given in the _speak_ tag `[nao:findfirstwall]` which is responsible to execute the function.

In the section IV.2.2 Post-executing Functions on Client Sidewe have seen hoe to manipulate the `nao_output.py` file which detects the code-word and executes the corresponding function. In the code of python we add these lines to execute the function.
elif command == "findfirstwall":
    x = findReqWall("FirstWall", "front")

Note that the function is call to find "FirstWall" and in the "front" direction. The direction can be asked in the dialogue rule also, but to keep simple the default direction is given as "front".

In the next section, we will see how the robot will act on the next dialogue rule "move_to_wall" and how he follows the QR code.

V.4.2 Moving Towards the found QR Code

In the last section the robot has searched the QR code asked by rotating his head. Then he executed the function "x = findReqWall("FirstWall", "front")" giving output 'x' and putting the dialogue rule "move_to_wall" in queue, the dialogue rule is defined below:

```xml
<rule descr="dialogue rule to move to wall">
    <conditions>
        <input contains="move_to_wall"/>
    </conditions>
    <actions>
        <speak text="Should I move to the wall."/>
    </actions>
</rule>

<!-- Terminal rule -->
<rule terminal="TRUE" descr="okay to choice" example="okay, you can go">
    <conditions>
        <granule concept="okay"/>
    </conditions>
    <actions>
        <speak text="OK, let's go to the wall.[nao:movetowall]"/>
    </actions>
</rule>

<!-- Non-terminal rule -->
<rule descr="no direction" example="hello">
    <conditions>
        <nogranule offer="direction"/>
    </conditions>
    <actions>
        <speak text="Please answer me: right or left ? [timeout:5]"/>
    </actions>
</rule>
</rule>
```
The conversation continues:

*Human:* Can you find the first wall?

(Or Find first wall)

*Robot:* Yeah sure, I will search the first wall around me.

(The robot starts rotating head and searches for first wall)

*Robot:* I found the first wall.

(This is the queueing of the "move_to_wall")

*Robot:* Should I move to the wall.

*Human:* Okay, you can move.

(This dialogue executes the function [nao:movetowall])

*Robot:* Okay, let’s go the wall.

The function corresponding to the [nao:movetowall] is given below:

```python
def moveToWall(x):
    global motionProxy
    # Turn whole body to the variable "x"
    motionProxy.moveTo(0.0, 0.0, x)
    # Move to this direction
    motionProxy.setWalkTargetVelocity(1.0, 0.0, 0.0, 1.0)
    # Obstacle detector to stop if it is 0.4m far
    detectObsStop(0.4)
    return 0
```

As, it is clear that the function \(x = \text{findReqWall}(\text{"FirstWall"}, \text{"front"})\)” will return a value ‘x’ which is a Yaw angle of the head is inputted to the function moveToWall(x). This makes the robot to first turn whole body towards the QR code and then walk toward it and obstacle detector is activated to avoid obstacle and also to ensure that the robot is reached to the QR coded wall. The function detectObsStop(minDist) is defined in can be found in Appendix 0.

In this way we can ask the robot to find wall as many QR codes are there in the environment. These QR code can be attached to the objects and the robot can be asked to follow the QR coded objects for example chairs, tables, cupboards etc.
V.5 Summary

In this chapter we have seen the specific task implementation that can be used to navigate Nao robot verbally. This gives that the robot can assist the human in the known environment, provided that the human user guiding verbally the robot to navigate. The implemented part used the QR code to identify the objects and walls. These codes are attached to the wall or objects and the robot can be asked to find specific things in the environment. The robot can search and move towards the code with verbal communication with the human. We have seen the detail, dialogue rules and python functions to control the robot by spoken dialogue system. There are many such functions that can be implemented with the help of this dialogue toolkit.

The aim was to show the ability of the YADTK dialogue toolkit to implement such tasks with ease of programming simple dialogue rules, reusing the granules and using the python function of the NAOqi Python SDK.
Concluding Remarks

This final chapter summarizes the main contributions of the thesis to conclude certainly a new approach in human-humanoid interaction to develop a natural language interface for the humanoid robot. This is a very brief review of the work on the theoretical and practical implementations of the spoken dialogue system and natural language interface. The advantages of rule-based spoken dialogue system with limitations. In addition, we also enumerate the research work that have not been directly addressed in this thesis, but can constitute important scope for future enhancement.

VI.1 Conclusions

There have been many research and work in the field of the human-humanoid interaction, but still it is obvious that the development of the natural language interface for humanoid robot is a tricky way. The approach of the natural language processing plays a very important role in the development. The hybrid approach can be a solution, but a lot of focus has to give on the humanoid robots, as they are very dynamic bodies. Humanoid robots are a moving and walking creature that can perform many tasks. Considering the fact that the human-computer interactions has significant distinct nature than human-robot interactions, same for the humanoid robots.

The YADTK dialogue toolkit was first developed for the human-computer interactions, but the work has shown that the toolkit is also well adapted to the human-robot interactions. The robots are very different than the computer machines. The computer machines are static in nature, whereas the robots are very dynamic and anthropoid robots are designed to resemble humans and mimic their behaviour.

In a technical manner, YADTK, the dialogue toolkit provided a lot of flexibility to program the humanoid robot, as given in the chapter IV Integration of Nao with YADTK. The two
simple XML formalisms, YADEML and YASPML, used to define the knowledge of the conversations, are very understandable and extensible. These are maintainable and reusable, provided that they are well designed, organized and documented. The dialogue rules and granules are simple to write and understand, even if one doesn’t know linguistic very well, as it is totally rule-based approach. The granules defined for the dialogue system are reusable as we said before, and the dialogue system we build in the work was less than 50, that covered the communication of greetings, daily routine tasks, verbal aid for navigation, etc.

The knowledge and concepts developed for the robot can be used for any machine, provided that the robot and other machines or the robots have same compliance. Simply, the knowledge is transferable within the circumstances of the dialogue toolkit.

In the work we also discovered a most important application of this dialogue toolkit for the humanoid robot. As we discussed in the introductory section of chapter III Platforms, that we are developing not only a NLSI but also NLTI. The natural language text interface allows us to monitor the robot from remote area and controller asking the robot to perform particular tasks in a natural language and important is that the robot can answer in the same, natural language.

**VI.2 Future Scope**

In this work, a lot of importance is given to the verbal communication, but at the same time we tried to develop some sort of non-verbal communications, by using tactile sensor, proximity sensors, etc. There is a lot of scope to enhance the features to increase the capability of the robot, to express gestures, to show emotional expressions, etc.

As we have seen that the robot has very well adapted the task implemented in the last chapter V Navigating the Nao with Verbal Aid, to follow the QR codes in the environment. But we limit the function just my rotating head and finding the required code. This sometimes gives errors in navigation. There is need to use proper vision based detection technique for the exact direction of the code found in the environment. Also, there is a scope to identify such a daily basis problems and build a default knowledge for the humanoid robots.
Bibliography


Appendices

In this section, Appendices, the codes and some extra concept are written to avoid multiple appearance of similar things. The data of this appendix can be referred from the corresponding sections of thesis.

Appendix 0

```python
# Function to detect the obstacle minimum distance (minDist)
def detectObsStop(minDist):
    while True:
        # Subscribe to sonars, this will launch sonars
        # (at hardware level) and start data acquisition.
        sonarProxy.subscribe("myApplication")
        # Now you can retrieve sonar data from ALMemory.
        # Get sonar left and right first echo
        # (distance in meters to the first obstacle).
        l = memoryProxy.getData("Device/SubDeviceList/US/Left/Sensor/Value")
        r = memoryProxy.getData("Device/SubDeviceList/US/Right/Sensor/Value")
        # Unsubscribe from sonars, this will stop sonars
        # (at hardware level)
        sonarProxy.unsubscribe("myApplication")
        # One can set minimum distance from 0.3m to 2.5m
        # Recommended distance is about 0.4m-1.0m
        if l < minDist or r < minDist:
            MOTION.setWalkTargetVelocity(0.0, 0.0, 0.0, 0.0)
            # talk("Oh there is obstacle")
            # queue.put("right_left_question")
            break
        time.sleep(0.05)
```
Appendix 1

<!-- Initiative Nested Rule (IR) -->
<rule descr="request without requestable" example="can I have">
  <conditions>
    <granule concept="request" ident="?id1">
      <nogranule/>
    </granule>
  </conditions>
  <actions>
    <speak text="I did not understand well what you ask me."
    test="(=(counter-value reiteration)1)"/>
    <speak text="What are you looking for exactly?"
    test="(=(counter-value reiteration)2)"/>
  </actions>
</rule>

<!-- Nested Terminal Rule (NTR) -->
<rule terminal="TRUE" descr="identifies a requestable" example="a ticket">
  <conditions>
    <granule offer="requestable" ident="?id2"/>
  </conditions>
  <actions>
    <speak text="(rephrase ?id2), Ok."
    test="(=(counter-value reiteration)1)"/>
  </actions>
</rule>

<!-- Nested Non Terminal Rule (NR) -->
<rule descr="not a requestable then reiterates">
  <conditions>
    <nogranule offer="requestable"/>
    <verify test="(< (counter-value reiteration) 2)"/>
  </conditions>
  <actions>
    <increase counter="reiteration"/>
    <speak text="Excuse me but you asked me something?" test="(=(counter-value reiteration)1)"/>
    <speak text="What do you want? Please answer me..." test="(=(counter-value reiteration)2)"/>
  </actions>
</rule>

<!-- Nested Terminal Rule (NTR) -->
<rule terminal="TRUE" descr="too much reiteration">
  <conditions>
    <nogranule offer="requestable"/>
    <verify test="(= (counter-value reiteration) 2)"/>
  </conditions>
  <actions>
    <speak text="Sorry but we definitively do not understand ourselves"
    test="(=(counter-value reiteration)1)"/>
    <remove counter="reiteration"/>
  </actions>
</rule>
</rule>
Appendix 2

The function `frange()` is defined to iterate the floating values. It is a simple function, takes input limits ‘start’ and ‘stop’ and ‘step’ size, and outputs the related values from lower limit to higher limit with respect to step size.

```python
def frange(start, stop, step):
    i = start
    while i < stop:
        yield i
        i += step
```

Appendix 3

```xml
<declare-feature id="execution"/>
<declare-feature id="order"/>

<granule concept="ordering" offers="execution">
    <dependency id="A2" expected="order"/>
    <syntax pattern="can you (A2)"/>
    <syntax pattern="can (A2)"/>
</granule>

<granule concept="findfirstwall" offers="order">
    <syntax pattern="find/search (a) first wall"/>
</granule>
```
Appendix 4

```xml
<rule descr="nao_initiative_on_silence">
  <conditions>
    <input contains="nao_initiative_on_silence"/>
  </conditions>
  <actions>
    <speak text="Hey, is there anybody? Speak with me. [timeout:10]"/>
  </actions>
</rule>

<!-- Terminal rule -->
<rule terminal="TRUE" descr="okay" example="Oh, what’s up? I was working">
  <conditions>
    <granule concept="okay"/>
  </conditions>
  <actions>
    <speak text="Okay, I thought you forgot me."/>
  </actions>
</rule>

<!-- Non-terminal rule -->
<rule descr="no direction" example="hello">
  <conditions>
    <nogranule offer="direction"/>
  </conditions>
  <actions>
    <speak text="Oh, I think nobody is there. I must sit down, my motor gets harder. [nao:relax]"/>
  </actions>
</rule>
</rule>
```