## 

As the robot has a humanoid shape it is possible to use several non-verbal social cues. The most important but sometimes overlooked non-verbal communication is approaching. A second important social cue is eye contact as it is considered to play an important role in many aspects of human-human communication such as timing and joint attention. Third, a substantial part of human-robot interaction involves understanding the robot’s intentions. To that end the robot thinks out loud and produces human-like movements. Also, human speech is usually accompanied by gesticulation, small gestures that improve understanding. Finally, as a robot is an electronic device, its non-human features like the eye LEDs are used to provide feedback to the user. In PT2 all these features are present and captured in the newly developed dialog management system.

### The Dialog Management System

In prototype 1 many features like thinking out loud, using the eye LEDs and gesticulation were tested. Given the positive formative evaluation it is safe to say that this functionality worked as expected. For prototype 2 we developed a way to combine dialog with gestures and eye LED signals. This was done in such a way that the content of the dialog is separated from the code that implements it.

#### Description of the DialogStateMachine Class

The core component is a python program DialogStateMachine.py that implements an SM\_Dialog class. It is implemented as a state machine for the SMACH package of the ROS system (<http://www.ros.org/wiki/smach> ). When initialized the class contains a link to a dialog file that defines the content and three states i.e. the Start state, the DoQuestion state and the GetAnswer. A graphical overview of the DialogStateMachine is shown in Figure 2.

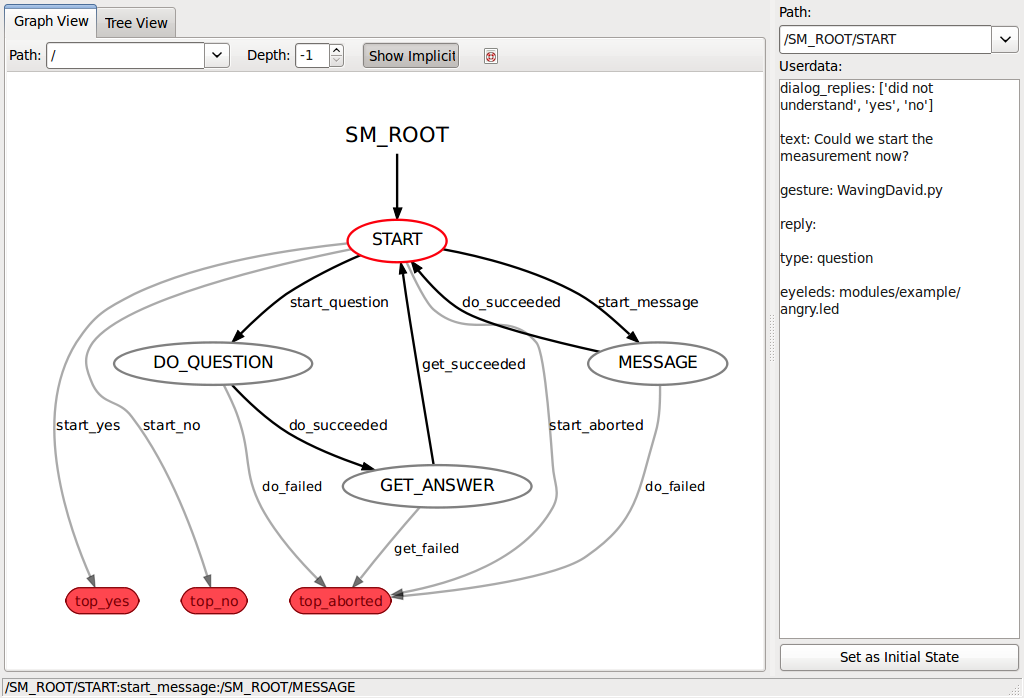


Figure 2: Structure of the DialogStateMachine class in action as visualized with the ROS Smach viewer package.

The Start state initializes the link with the robot using Aldebaran’s naoqi SDK and it reads the dialog file that is passed to the constructor of this class during initialization. The dialog file contains the texts of the questions to ask and the messages to deliver. It also contains the logical sequence of how to proceed from one question/message to another based on the answers that are given. Furthermore, it contains the filenames of files containing the description of accompanying gestures and facial LED patterns. All files are simple text files and can be edited without changing the code of the KSERA system. A detailed description of the dialog files and the related gestures files and eye LED pattern files is given in the appendix 11.2 – 11.5.

Once the initialization is done, the Start state reads the next entry of the dialog file and determines whether a question should be asked or a message delivered or that the state machine should end.

The Doquestion and Message state are mostly the same. They take care of sending the proper commands to the robot so that the speech, gestures and LED patterns are played back in the correct way. The message state either succeeds and goes back to the start state or it fails and the state machine ends. If a question was asked, the robot first needs to listen for a reply, so the DoQuestion state proceeds to the GetAnswer state. The gesture files are python scripts that contain the names, times and values of the joint angles of the NAO robot. They are most easily generated using the Choregraphe graphical programming tool provided by Aldebaran robotics (see appendix). The LED pattern files are simple text files containing a list of LED colours, display durations and frequencies for a number of predefined LED groups (see appendix). The speech, gestures and LED patterns are played back in parallel, starting at the same time. To coordinate these behaviours the dialog files must be field tested so that the proper text is spoken while the gestures and LED patterns are displayed.

The GetAnswer state invokes the speech recognition engine of the KSERA system and waits for a reply. If the answer was not intelligible the question is repeated, but no more than three times. Optionally, keyboard input can be given to override the speech recognition. This way the proper functioning of the system is guaranteed during the field trials as an experimenter can give the correct answer if the speech recognition is not recognizing properly. As soon as a proper answer is given the state machine proceeds to the Start state with the reply.

#### Using the DialogStateMachine Class

To use the DialogStateMachine a connection to the robot and the speech recognition system it needed. In addition the dialog files containing the content of the dialogs is needs. Suppose there are two dialog files dialog1.csv and dialog2.csv, the DialogStateMachine class is invoked as follows:

#import relevant stuff

from DialogStateMachine import SMDialog

from readconfigfile import ReadConfigKey

#Read nao ip

mykey=ReadConfigKey('IP\_NAO')

mynao\_ip=mykey[1]

#Create any number of instances of the state machine class SMDialog() and extract the statemachine sm\_top:

sm\_dialog1=SMDialog("dialog1.csv", mynao\_ip).sm\_top

sm\_dialog2=SMDialog("dialog2.csv", mynao\_ip).sm\_top

# add them to your own state machine

smach.StateMachine.add('DIALOG\_1', sm\_dialog1,

transitions={'top\_yes': 'next\_state\_if\_yes',

'top\_no': 'next\_state\_if\_no',

'top\_aborted':'next\_state\_if\_aborted'})

#### Summary

The DialogStateMachine class is a very powerful dialog management system that is compact and can be reused in different contexts as it separates content from implementation. It also combines gestures, LED patterns and speech in a relatively straightforward but flexible way, so that it is simple to build new content.

## The Dialog files

The most important feature of the DialogStateMachine.py is that it completely separates the content and the logic of a dialogue from the code itself. In other words what the robot says and how it carries out a conversation, what gestures it shows and which eye LED patterns accompany it is completely specified in the dialog files.

There are three types of files:

The dialog files, which are .csv files (semi-colon separated values in our case) that reside in david/lib/dialogs

The gesture files, which are python scripts containing a movement pattern. They reside in david/lib/gestures

The LED pattern files, which are .csv (semi-colon separated values in our case) residing in david/lib/led

### Dialog .csv files

The dialog .csv files contain three sections:

dialog\_keys

dialog\_logic

message\_logic

#### Section dialog\_keys:

Each dialog .csv file must have a row in which the first column is **dialog\_keys**. (Text in other columns is ignored and used only for human reading in the form of headers.)

The rows below the section header dialog\_keys are interpreted as containing questions and messages. The format is as follows:

Contains text with the name to identify the question or message. The convention is to start with ‘key’ followed by word(s) reflecting the content e.g. ‘keyAskMeasurement’. Do not use spaces.

Contains the type of the utterance: ‘message’ or ‘question’ depending on whether the robot should listen for a response or not.

Name of the gesture file, which is searched for in PYTHONPATH. It is possible to include subdirectories in the name, by default it looks in the /gestures subdirectories in david’s rosnode david/lib/gestures

Name of the LED filename, which reside in david/lib/led.

The content of the message or the question e.g. “Do you want to do the OxyPulsometer measurement now?”

– Column N: all subsequent columns are interpreted as text alternatives for the same question or message. These are optional.

Note that the 1st key in this list is considered the initial question!

#### Section dialog\_logic

The dialog\_logic section specifies what should happen after a question has been asked. Depending on the answer the dialogue should either continue with another question, a message or just end.

The dialog .csv file should contain a row with **dialog\_logic** in the first column. As before other columns in this row are ignored. They are used only for human reading i.e. headers.

The rows below the section header dialog\_logic are interpreted as containing a logical mapping between keys of the questions and messages. Each line specifies for a given key which answer is mapped on which follow-up keys. The possible replies depend on the speech recognition. Currently, supported replies are ‘yes’, ‘no’, ‘nao stop’, and ‘did not understand’. If a key is missing the program will use the default mapping to exit the state machine (see DialogStateMachine). The keys that are mapped to should be defined in the section **dialog\_keys**. In addition the special keys **endYes** and **endNo** can be used. The specify with which state the DialogStateMachine ends.

The format of the dialog\_logic section is given below:

Contains the text specifying the name of the key that identifies the question (messages do not have replies).

Text containing reply1 (currently only ‘yes’, ‘no’, ‘nao stop’, and ‘did not understand’) are supported by the speech recognition.

Text containing the name of the key for the next question or message in case reply1 is given

… and so on for the other replies.

##### Notes

It is no problem to add more and other replies, but they are currently not recognized.

If a reply is given but not found than the DialogStateMachine ends with the default mapping. All unrecognized speech is currently mapped to ‘did not understand’

If speech is unrecognized (ASRsphinx output is ‘did not understand’), the question will be repeated three times before it goes to the exit state given by the dialog\_logic key.

##### Example

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| dialog\_logic | reply1 | next\_key1 | reply2 | next\_key2 | reply3 | next\_key3 |
| keyConfirm | yes | keyDeny | no | keyAccept | did not understand | keyDeny |

After question “keyConfirm” has been asked, a reply of “yes” will take the conversation to message “keyDeny” , “no” will go to message “keyAccept” and “did not understand” will go to “keyDeny”, but only after asking this same question 3 times in a row.

#### Section message\_logic

The message\_logic section specifies what should happen after a message has been said. In this case there is no answer, but sometimes one still needs to specify where to go to next, or in which state to end.

The dialog .csv file should contain a row with **message\_logic** in the first column. As before other columns in this row are ignored. They are used only for human reading i.e. headers.

The rows below the section header message\_logic are interpreted as containing a logical mapping between keys of the questions and messages. Each line specifies for a given messge key the follow-up keys. If a key is missing the program will use the default mapping to exit the state machine (the default is top\_yes, see DialogStateMachine). The keys that are being mapped to should be defined in the section **dialog\_keys**. In addition the special keys **endYes** and **endNo** can be used. The specify with which state the DialogStateMachine ends. If multiple keys are given the DialogStateMachine will randomly pick one of them.

The format of the message\_logic section is given below:

Contains the text specifying the name of the key that identifies the message.

Text containing the name of the key of the next utterence.

- Column N: Optionally you can specify more than one alternative. The DialogStateMachine will randomly pick one of them.

#### Example

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| dialog\_logic | reply1 | next\_key1 | reply2 | next\_key2 | reply3 | next\_key3 |
| keyConfirm | yes | keyDeny | no | keyAccept | did not understand | keyDeny |

After question “keyConfirm” has been asked, a reply of “yes” will take the conversation to message “keyDeny” , “no” will go to message “keyAccept” and “did not understand” will go to “keyDeny”, but only after asking this same question 3 times in a row.

Here is the first dialog file ask\_measurement.csv of IF1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **dialog\_keys** | **type** | **gesture file** | **LED file** | **Alternative 1** | **Alternative 2** | **Alternative 3** | **Alternative 4** |
| keyMeasurement | question | QuestionGesture.py | QuestionLed\_5.csv | Could we start the measurement now? | Should we perform a measurement now? | Do you have time to perform a measurement now? |  |
| keyConfirm | question | QuestionGesture.py | QuestionLed\_5.csv | Are you sure? | Really not? | But it would be good for you, perhaps you would still like to do it? |  |
| keyAccept | message | CommandGesture.py | HappyLed\_5.csv | Great! Please do the measurement with the pulseoximeter. |  |  |  |
| keyDeny | message | SadGesture.py | SadLed\_5.csv | OK, then not today! | That is a pity. I will ask you again tomorrow! | It is a pity, it would have been nice. | How unfortunate. I was looking forward to it! |
| **dialog\_logic** | **reply 1** | **next\_key 1** | **reply 2** | **next\_key 2** | **reply 3** | **next\_key 3** |  |
| keyMeasurement | yes | keyAccept | no | keyConfirm | did not understand | keyDeny |  |
| keyConfirm | yes | keyDeny | no | keyAccept | did not understand | keyDeny |  |
| **message\_logic** | **Alternative 1** | **Alternative 2** | **Alternative 3** | **Alternative 4** |  |  |  |
| keyAccept | endYes |  |  |  |  |  |  |
| keyDeny | endNo |  |  |  |  |  |  |

### Gesture files

Gesture files are made using Choregraphe. With choregraphe it is easy to record movements, if necessary step by step, with the real robot. You can then simply export python code to the clipboard from the context menu (see Choregraphe manual for details). Pasting and save it into a text file and you have created a gesture file. Make sure you use .py or .ges as extention.

The gesture file contains three lists:

names – the names of the joints of NAO

times – the times at which the joint angle values should occur

keys – the joint angles for each of the joints specified in names

Notes:

The gesture files work with the RunMovement function in David’s david/lib/nao.py.

They assume the RunMovement function parameter, to\_start\_position is set to True, that is, the gesture files do not return NAO to the starting position.

#### Currently Implemented

AttentionGesture.py - NAO raises right arm with hand open (trying to imitate the human one finger up "just a moment" gesture)

CommandGesture.py - NAO twists his right arm and shows the palm of his right hand to the user

DontKnowGesture.py - NAO shrugs with palms up

HappinessGesture.py - NAO nods head up and down once

QuestionGesture.py - NAO twists both arms and shows the palms of his hands to the user

UnhappinessGesture.py - NAO shakes head right to left once

### LED Pattern Files

The LED pattern files are semi-colon delimited .csv files with a header line. Each line contains the groupname of the LEDs, the colour in r, g, b, coordinates, the frequency in Hz, and the duration. If the frequency field contains 0 the LEDs will be continuously on. For any given value of the frequency the LED group will flash on and off for the duration in seconds.

By providing multiple lines, quite complex LED patterns can be made. Here is the Happy LED pattern.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| groupname(see help.txt) | r | g | b | frequency(0 = continuously on) | duration (s) |
| FaceLeds | 200 | 120 | 0 | 0 | 2 |
| FaceLeds | 0 | 0 | 0 | 0 | 0.5 |
| FaceLeds | 200 | 120 | 0 | 0 | 2 |
| FaceLeds | 0 | 0 | 0 | 0 | 0.5 |

For the group names use one of the following groups:

['AllLeds', 'AllLedsBlue', 'AllLedsGreen', 'AllLedsRed', 'BrainLeds', 'BrainLedsBack', 'BrainLedsFront', 'BrainLedsLeft', 'BrainLedsMiddle', 'BrainLedsRight', 'ChestLeds', 'EarLeds', 'FaceLed0', 'FaceLed1', 'FaceLed2', 'FaceLed3', 'FaceLed4', 'FaceLed5', 'FaceLed6', 'FaceLed7', 'FaceLedLeft0', 'FaceLedLeft1', 'FaceLedLeft2', 'FaceLedLeft3', 'FaceLedLeft4', 'FaceLedLeft5', 'FaceLedLeft6', 'FaceLedLeft7', 'FaceLedRight0', 'FaceLedRight1', 'FaceLedRight2', 'FaceLedRight3', 'FaceLedRight4', 'FaceLedRight5', 'FaceLedRight6', 'FaceLedRight7', 'FaceLeds', 'FaceLedsBottom', 'FaceLedsExternal', 'FaceLedsInternal', 'FaceLedsLeftBottom', 'FaceLedsLeftExternal', 'FaceLedsLeftInternal', 'FaceLedsLeftTop', 'FaceLedsRightBottom', 'FaceLedsRightExternal', 'FaceLedsRightInternal', 'FaceLedsRightTop', 'FaceLedsTop', 'FeetLeds', 'LeftEarLeds', 'LeftEarLedsBack', 'LeftEarLedsEven', 'LeftEarLedsFront', 'LeftEarLedsOdd', 'LeftFaceLeds', 'LeftFaceLedsBlue', 'LeftFaceLedsGreen', 'LeftFaceLedsRed', 'LeftFootLeds', 'RightEarLeds', 'RightEarLedsBack', 'RightEarLedsEven', 'RightEarLedsFront', 'RightEarLedsOdd', 'RightFaceLeds', 'RightFaceLedsBlue', 'RightFaceLedsGreen', 'RightFaceLedsRed', 'RightFootLeds']