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:

1. 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.
2. Contains the type of the utterance: ‘message’ or ‘question’ depending on whether the robot should listen for a response or not.
3. 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
4. Name of the LED filename, which reside in david/lib/led.
5. The content of the message or the question e.g. “Do you want to do the OxyPulsometer measurement now?”
6. – 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:

1. Contains the text specifying the name of the key that identifies the question (messages do not have replies).
2. Text containing reply1 (currently only ‘yes’, ‘no’, ‘nao stop’, and ‘did not understand’) are supported by the speech recognition.
3. Text containing the name of the key for the next question or message in case reply1 is given
4. … 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:

1. Contains the text specifying the name of the key that identifies the message.
2. Text containing the name of the key of the next utterence.
3. - 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.

## Example

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']