DialBB Document

Release v0.8.0

Nov 09, 2024

CONTENTS:

CHAPTER

ONE

INTRODUCTION

DialBB (Dialogue System Development Framework with Building Blocks) is a framework for building dialogue systems.

Dialogue systems are constructed by integrating various technologies in the information technology field. Using this framework, we aim to enable people with little knowledge of dialogue system technology and little experience in information system development to construct dialogue systems and learn various information technologies. We also aim to make DialBB easy to understand the architecture, highly extensible, and easy to read code, so that it can be used as a teaching material for programming and system development education.

Please see the README for instructions on how to install DialBB and how to run the sample application.

CHAPTER

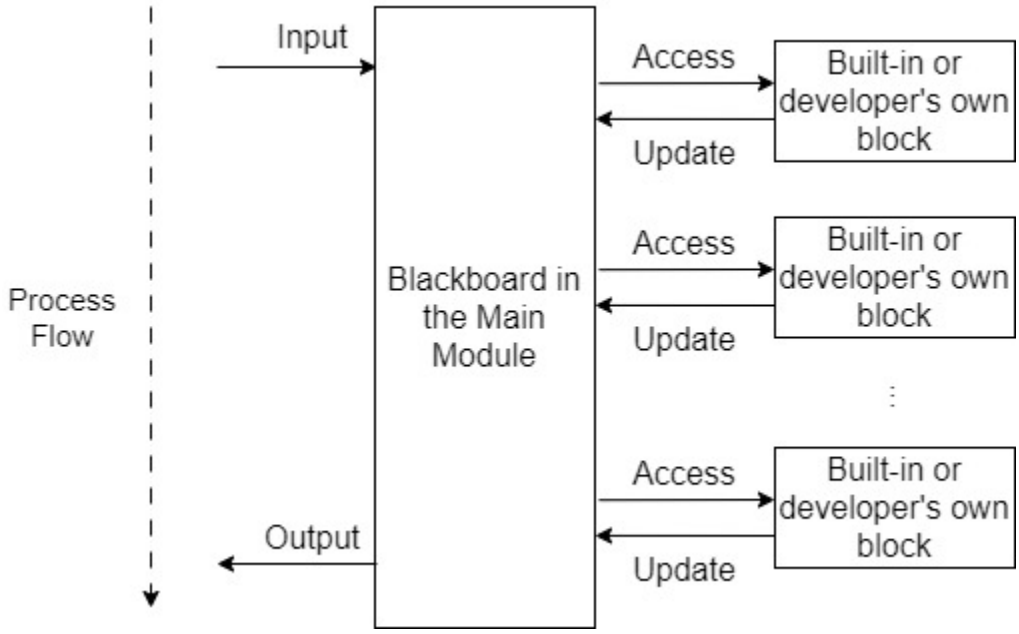
TWO

DIALBB OVERVIEW

As mentioned in the introduction, DialBB is a framework for building dialogue systems.

A framework does not stand alone as an application, but forms an application by providing data and additional programs.

The basic architecture of a DialBB application is shown below.



The main module creates and returns a system utterance by making modules called blocks sequentially process the data (including user utterances) inputted at each turn of the dialog. The inputted is data copied to data called blackboard[1] in the main block. Each block takes some of the elements of the blackboard and returns data in dictionary format. The returned data is added to the blackboard. If an element with the same key already exists in the blackboard, it is overwritten.

The type of block to be used is specified in the configuration file. Blocks can be either blocks provided by DialBB (built-in blocks) or blocks created by the application developer.

The configuration file also specifies what data the main module sends to and receives from each block.

Details are explained in the "*Framework Specifications*" section.

[1] Before ver. 0.2, it was called payload.

CHAPTER

THREE

SAMPLE APPLICATIONS

3.1 Sample Applications Included in DialBB

3.1.1 Parroting application

This is an application that just parrots back and forth. No built-in block classes are used.

It is located in `sample_apps/parrot`.

3.1.2 Snips+STN Applications

Applications using the following blocks.

- English Application
  - *Simple Canonicalizer (Simple String Canonicalizer Block)*
  - *Whitespace Tokenizer (Whitespace-based Word Segmentation Block)*
  - *Snips Understander (Language Understanding Block using Snips)*
  - *STN Manager (State Transition Network-based Dialogue Management Block)*
- Japanese Application
  - Japanese Canonicalizer
  - Sudachi Tokenizer
  - *Snips Understander (Language Understanding Block using Snips)*
  - *STN Manager (State Transition Network-based Dialogue Management Block)*

The English version is located in `sample_apps/network_en/` and the Japanese version is located in `sample_apps/network_ja/`.

3.1.3 Experimental Application

This application is based on language understanding based on ChatGPT and dialogue management based on state transistion networks, and includes examples of various functions of the built-in blocks. It uses the following built-in blocks. (from v0.7, ChatGPT language undderstanding instead of Snips language understanding)

- English Application
  - Simple Canonicalizer
  - *ChatGPT Understander (Language Understanding Block using ChatGPT)*
  - *spaCy-Based NER (Named Entity Recognizer Block using spaCy)*
  - *STN Manager (State Transition Network-based Dialogue Management Block)*
- Japanese Application
  - Japanese Canonicalizer
  - *ChatGPT Understander (Language Understanding Block using ChatGPT)*
  - *spaCy-Based NER (Named Entity Recognizer Block using spaCy)*
  - *STN Manager (State Transition Network-based Dialogue Management Block)*

It is located in `sample_apps/lab_app_ja/` (Japanese) and `sample_apps/lab_app_en/` (English).

3.1.4 ChatGPT Dialogue Application

It uses the following built-in blocks to engage in dialogue using OpenAI's ChatGPT.

- *ChatGPT Dialogue (ChatGPT-based Dialogue Block)*

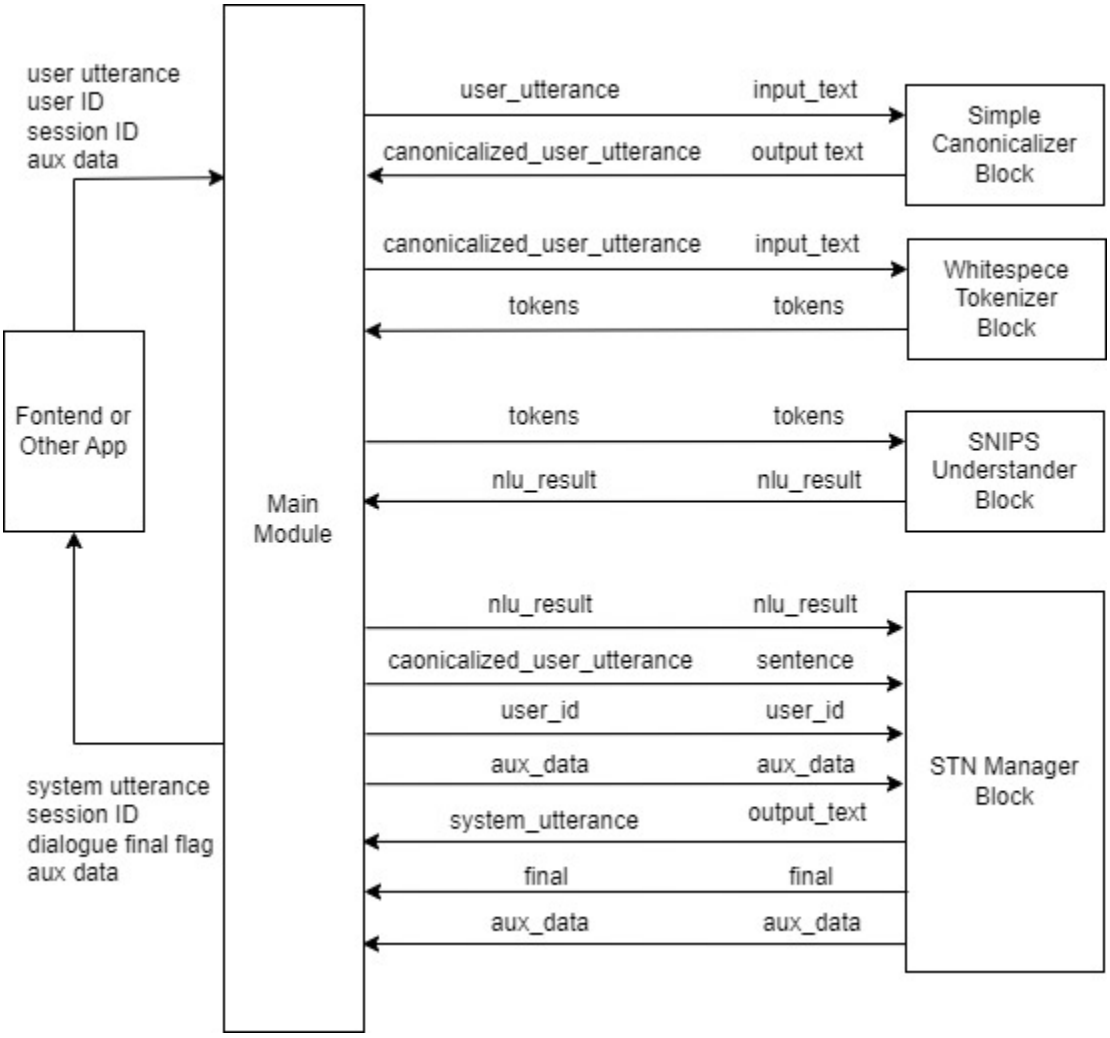
It is located in `sample_apps/chatgpt/`.

3.2 Explanation of Snips+STN Application

This section describes the structure of a DialBB application through the English Snips+STN Application.

3.2.1 System Architecture

Below is the system architecture of the application.



This application uses the following built-in blocks. The details of these built-in blocks are described in "*Built-in Block Classes*".

- Simple Canonicalizer: Normalizes user input text (uppercase -> lowercase, etc.)
- Whitespace Tokenizer: Splits normalized user input into tokens based on whitespaces.
- Snips Understander: Performs language understanding, using Snips_NLU to determine user utterance types (also called intents) and extract slots.
- STN Manager: Performs dialogue management and language generation. It uses a state transition network and outputs system utterances.

The symbols on the arrows connecting the main module and the blocks are the keys on the blackboard of the main module on the left side and the keys on the input/output of the blocks on the right side.

3.2.2 Files Comprising the Application

The files comprising this application are located in the directory `sample_apps/network_en`. By modifying these files, you can see how the application will change. By making significant changes to the files, you can create a completely different dialogue system.

`sample_apps/network_en` includes the following files.

- `config.yml`

  This is a configuration file that defines the application. It specifies information such as what blocks to use and the files to be loaded by each block. The format of this file is described in detail in the "*Configuration*" section.

- `sample-knowledge-en.xlsx`

  This describes the knowledge used in the Snips Understander and STN Manager blocks.

- `scenario_functions.py`

  This defines functions used in the STN Manager block.

- `dictionary_functions.py`

  This contains examples of defining dictionary for Snips Understander with functions, not Excel.

- `test_inputs.txt`

  Test scenarios used in system testing.

3.2.3 Snips Understander Block

Language understanding results

The Snips Understander block analyzes input utterance and outputs language understanding results. Each result consists of a type and a set of slots. For example, the language understanding result of "I like chicken salad sandwich" is as follows.

```
{
  "type": "tell-like-specific-sandwich",
  "slots": {
    "favarite_sandwich": "chcken salad sandwich"
  }
}
```

The type is `"tell-like-specific-sandwich"` and the value of the `"favarite_sandwich"` slot is `"chicken salad sandwich"`. It is possible to have multiple slots.

Language understanding knowledge

The knowledge for language understanding used by the Snips Understander block is written in `sample-knowledge-ja.xlsx`.

The language understanding knowledge consists of the following four sheets.

Sheet name	Contents
utterances	Examples of utterance for each type
slots	Relationship between slots and entities
entities	Entity information
dictionary	Dictionary entries and sysnonyms for each entry

For more information on these, see "*Language Understanding Knowledge*."

Training data for Snips

When the application is launched, the above knowledge is converted into training data for Snips and a model is created.

The training data for Snips is `_training_data.json` in the application directory. You can check if the conversion is successful by looking at this file.

3.2.4 STN Manager Block

The dialogue management knowledge (scenario) is the `scenario` sheet in the `sample-knowledge-en.xlsx` file.

For details on how to write this sheet, please refer to "*Dialogue Management Knowledge Description*".

If GraphViz is installed, the application outputs an image file (_) of the state transition network generated from the scenario file when the application is launched. The following is the state transition network of the application.

Among the functions used in the conditions of transitions and actions that are executed after transitions, those that are not built-in are defined in `scenario_functions.py`.

CHAPTER

FOUR

FRAMEWORK SPECIFICATIONS

This section describes the specifications of DialBB as a framework.

We assume that the reader has knowledge of Python programming.

4.1 Input and Output

The main module of DialBB has the class API (method call), which accepts user utterance and auxiliary information in JSON format and returns system utterance and auxiliary information in JSON format.

The main module works by calling blocks in sequence. Each block receives data formatted in JSON (Python dictionary type) and returns the data in JSON format.

The class and input/output specifications of each block are specified in the configuration file for each application.

4.1.1 The DialogueProcessor Class

The application is built by creating an object of class `dialbb.main.DialogueProcessor`

This is done by the following procedure.

- Add the DialBB directory to the `PYTHONPATH` environment variable.

```
export PYTHONPATH=<DialBB directory>:$PYTHONPATH
```

- In the application that uses DialBB, use the following DialogueProcessor and calls process method.

```
from dialbb.main import DialogueProcessor
dialogue_processor = DialogueProcessor(<configuration file> <additional␣
↪configuration>)
response = dialogue_processor.process(<request>, initial=True)  # at the start of a␣
↪dialogue session
response = dialogue_processor.process(<request>) # when session continues
```

<additional configuration> is data in dictionary form, where keys must be a string, such as

```
{
  "<key1>": <value1>,
  "<key2>": <value2>,
  ...
}
```

This is used in addition to the data read from the configuration file. If the same key is used in the configuration file and in the additional configuration, the value of the additional configuration is used.

<request> and response are dictionary type data, described below.

Note that DialogueProcessor.process is **not** thread safe.

4.1.2 Request

At the start of the session

JSON in the following form.

```
{
  "user_id": <user id: string>,
  "aux_data": <auxiliary data: object (types of values are arbitrary)>}
}
```

- user_id is mandatory and aux_data is optional
- <user id> is a unique ID for a user. This is used for remembering the contents of previous interactions when the same user interacts with the application multiple times.
- <auxiliary data> is used to send client status to the application. It is an JSON object and its contents are decided on an application-by-application basis.

After the session starts

JSON in the following form.

```
{
  "user_id": <user id: string>,
  "session_id": <session id: string>,
  "user_utterance": <user utterance string: string>,
  "aux_data": <auxiliary data: object (types of values are arbitrary)>
}
```

- user_id, session_id, and user_utterance are mandatory, and aux_data is optional.
- <session id> is the session ID included in the responses.
- <user utterance string> is the utterance made by the user.

4.1.3 Response

```
{
  "session_id":<session id: string>,
  "system_utterance": <system utterance string: string>,
  "user_id":<user id: string>,
  "final": <end-of-dialogue flag: bool>
  "aux_data": <auxiliary data: object (types of values are arbitrary)>
}
```

- <session id> is the ID of the dialog session. A new session ID is generated when new session starts.

- <system utterance string> is the utterance of the system.
- <user id> is the ID of the user sent in the request.
- <end-of-dialog flag> is a boolean value indicating whether the dialog has ended or not.
- <auxiliary data> is data that the application sends to the client. It is used to send information such as server status.

4.2 WebAPI

Applications can also be accessed via WebAPI.

4.2.1 Server Startup

Set the PYTHONPATH environment variable.

```
export PYTHONPATH=<DialBB directory>:$PYTHONPATH
```

Start the server by specifying a configuration file.

```
python <DialBB directory>/run_server.py [--port <port>] <config file>
```

The default port number is 8080.

4.2.2 Connection from Client (At the Start of a Session)

- URI

  ```
  http://<server>:<port>/init
  ```

- Request header

  ```
  Content-Type: application/json
  ```

- Request body

  The data is in the same JSON format as the request in the case of the class API.

- Response

  The data is in the same JSON format as the response in the case of the class API.

4.2.3 Connection from Client (After the Session Started)

- URI

  ```
  http://<server>:<port>/dialogue
  ```

- request header

  ```
  Content-Type: application/json
  ```

- request body

  The data is in the JSON format as the request in the case of the class API.

- response

  The data is in the same JSON format as the response in the case of the class API.

4.3 Configuration

The configuration is data in dictionary format and is assumed to be provided with a yaml file.

Only the `blocks` element is required for configuration; the blocks element is a list of what each block specifies (this is called the block configuration) and has the following form

```
blocks:
  - <Block Configuration>
  - <Block Configuration>
  ...
  - <Block Configuration>
```

The following are the mandatory elements of each block configuration.

- `name`

  Name of the block. Used in the log.

- `block_class`

  The class name of the block. This should be written as a realtive path from a module search path (an element of `sys.path`. Paths set to `PYTHONPATH` environment variable are included in it).

  The directory containing the configuration files is automatically registered in the path (an element of `sys.path`) where the module is searched.

  Built-in classes should be specified in the form `dialbb.built-in_blocks.<module name>.<class name>`. Relative paths from `dialbb.builtin_blocks` are also allowed, but are deprecated.

- `input`

  This defines the input from the main module to the block. It is a dictionary type data, where keys are used for references within the block and values are used for references in the blackboard (data stored in the main module). For example, if the following is in a block configuration, then what can be referenced by `input['sentence']` in the block is `blackboard['canonicalized_user_utterance']` in the main module.

  ```
  input:
    sentence: canonicalized_user_utterance
  ```

  If the specified key is not in the blackboard, the corresponding element of the input becomes `None`.

- `output`

  Like `input`, it is data of dictionary type, where keys are used for references within the block and values are used for references on the blackboard. If the following is specified:

  ```
  output:
    output_text: system_utterance
  ```

  and if the output from the block is `output`, then the following process is performed.

```
blackboard['system_utterance'] = output['output_text']
```

If `blackboard` already has `system_utterance` as a key, the value is overwritten.

4.4 How to make your own blocks

Developers can create their own blocks.

The block class must be a descendant of `diabb.abstract_block.AbstractBlock`.

4.4.1 Methods to be Implemented

- `__init__(self, *args)`

  Constructor. It is defined as follows:

```
def __init__(self, *args):

    super().__init__(*args)

    <Process unique to this block>
```

- `process(self, input: Dict[str, Any], session_id: str = False) -> Dict[str, Any]`

  Processes input and returns output. The relationship between input, output and the main module's blackboard is defined by the configuration (see "*Configuration*"). `session_id` is a string passed from the main module that is unique for each dialog session.

4.4.2 Available Variables

- `self.config` (dictionary)

  This is a dictionary type data of the contents of the configuration. By referring to this data, it is possible to read in elements that have been added by the user.

- `self.block_config` (dictionary)

  The contents of the block configuration are dictionary type data. By referring to this data, it is possible to load elements that have been added independently.

- `self.name` (string)

  The name of the block as written in the configuration.

- `self.config_dir` (string)

  The directory containing the configuration files. It is sometimes called the application directory.

4.4.3 Available Methods

The following logging methods are available

- `log_debug(self, message: str, session_id: str="unknown")`

  Outputs debug-level logs to standard error output. `session_id` can be specified as a session ID to be included in the log.

- `log_info(self, message: str, session_id: str="unknown")`

  Outputs info level logs to standard error output.

- `log_warning(self, message: str, session_id: str="unknown")`

  Outputs warning-level logs to standard error output.

- `log_error(self, message: str, session_id: str="unknown")`

  Outputs error-level logs to standard error output.

4.5 Debug Mode

When the environment variable `DIALBB_DEBUG` is set to `yes` (case-insensitive) during Python startup, the program runs in debug mode. In this case, the value of `dialbb.main.DEBUG` is `True`. This value can also be referenced in blocks created by the application developer.

If `dialbb.main.DEBUG` is `True`, the logging level is set to debug; otherwise it is set to info.

4.6 Test Using Test Scenarios

The following commands can be used to test with test scenarios.

```
$ python dialbb/util/test.py <application configuration> \
  <test scenario> [--output <output file>]
```

The test scenario is a text file in the following format:

```
<session separation>
System: <system utterance>
User: <user utterance>
System: <system utterance>
User: <user utterance>
...
System: <system utterance>
User: <user utterance>
System: <system utterance>
<session separation>
<System: <system utterance>
User: <user utterance>
System: <system utterance>
User: <user utterance>
...
System: <system utterance>
User: <user utterance>
```

(continues on next page)

(continued from previous page)

```
System: <system utterance>
<session separation>
...
```

<session separation> is a string stareging with ----init.

The test script receives system utterance by inputting <user speech> to the application in turn. If the system utterances differ from the script's system utterances, a warning is issued. When the test is finished, the dialogues can be output in the same format as the test scenario, including the output system utterances. By comparing the test scenario with the output file, changes in responses can be examined.

CHAPTER

FIVE

BUILT-IN BLOCK CLASSES

Built-in block classes are block classes that are included in DialBB in advance.

Below, the explanation of the blocks that deal with only Japanese is omitted.

5.1 Simple Canonicalizer (Simple String Canonicalizer Block)

(`dialbb.builtin_blocks.preprocess.simple_canonicalizer.SimpleCanonicalizer`)

Canonicalizes user input sentences. The main target language is English.

5.1.1 Input/Output

- Input
  - `input_text`: Input string (string)
    - Example: "I like ramen".
- Output
  - `output_text`: string after normalization (string)
    - Example: "i like ramen".

5.1.2 Process Details

Performs the following processing on the input string.

- Deletes leading and tailing spaces.
- Replaces upper-case alphabetic characters with lower-case characters.
- Deletes line breaks.
- Converts a sequence of spaces into a single space.

5.2 Whitespace Tokenizer (Whitespace-based Word Segmentation Block)

(`dialbb.builtin_blocks.tokenization.whitespace_tokenizer.WhitespaceTokenizer`)

Splits input into words separated by spaces. This is mainly for English.

5.2.1 Input/Output

- input
  - `input_text`: Input string (string)
    - Example: "i like ramen".
- output (e.g. of dynamo)
  - `tokens`: list of tokens (list of strings)
    - Example: ['i','like','ramen'].
  - `tokens_with_indices`: List of token information (list of objects of class `dialbb.tokenization.abstract_tokenizer.TokenWIthIndices`). Each token information includes the indices of start and end points in the input string.

5.2.2 Process Details

Splits the input string canonicalized by the simple canonicalizer by whitespace.

5.3 Snips Understander (Language Understanding Block using Snips)

(`dialbb.builtin_blocks.understanding_with_snips.snips_understander.Understander`)

Determines the user utterance type (also called intent) and extracts the slots using Snips_NLU.

Performs language understanding in Japanese if the `language` element of the configuration is `ja`, and language understanding in English if it is `en`.

At startup, this block reads the knowledge for language understanding written in Excel, converts it into Snips training data, and builds the Snips model.

At runtime, it uses the created Snips model for language understanding.

5.3.1 Input/Output

- input
  - `tokens`: list of tokens (list of strings)
    - Example: `['I' 'like', 'chicken', 'salad' 'sandwiches']`.
- output
  - `nlu_result`: language understanding result (dict or list of dict)
    - If the parameter `num_candidates` of the block configuration described below is 1, the language understanding result is a dictionary type in the following format.

```
{
    "type": <user utterance type (intent)>,
    "slots": {<slot name>: <slot value>, ... , <slot name>: <slot value>}
}
```

The following is an example.

```
{
    "type": "tell-like-specific-sandwich",
    "slots": {"favorite-sandwich": "roast beef sandwich"}
}
```

    - If `num_candidates` is greater than 1, it is a list of multiple candidate comprehension results.

```
[{"type": <user utterance type (intent)>,
  "slots": {<slot name>: <slot value>, ... , <slot name>: <slot value>}},␣
↪...
 {"type": <user utterance type (intent)>,
  "slots": {<slot name>: <slot value>, ... , <slot name>: <slot value>}},␣
↪...
 .....]
```

5.3.2 Block Configuration Parameters

- `knowledge_file` (string)

  Specifies the Excel file that describes the knowledge. The file path must be relative to the directory where the configuration file is located.

- `function_definitions` (string)

  Specifies the name of the module that defines the dictionary functions (see *Dictionary function definitions by developers*). If there are multiple modules, connect them with `':'`. The module must be in the module search path. (The directory of the configuration file is in the module search path.)

- `flags_to_use` (list of strings)

  Specifies the flags to be used. If one of these values is written in the `flag` column of each sheet, it is read. If this parameter is not set, all rows are read.

- `canonicalizer`

  Specifies the canonicalization information to be performed when converting language comprehension knowledge to Snips training data.

  - `class`

    Specifies the class of the normalization block. Basically, the same normalization block used in the application is specified.

- `tokenizer`

  Specifies the tokenization information to be used when converting language understanding knowledge to Snips training data.

  - `class`

    Specifies the class of the tokenization block. Basically, the same tokenization block used in the application is specified.

- `num_candidates` (integer. Default value is 1)

  Specifies the maximum number of language understanding results (n for n-best).

- `knowledge_google_sheet` (hash)
  - This specfies information for using Google Sheet instead of Excel.
    * `sheet_id` (string)

      Google Sheet ID.

    * `key_file`(string)

      Specify the key file to access the Google Sheet API as a relative path from the configuration file directory.

5.3.3 Language Understanding Knowledge

Language understanding knowledge consists of the following four sheets.

sheet name	contents
utterances	examples of utterances by type
slots	relationship between slots and entities
entities	Information about entities
dictionary	dictionary entries and synonyms per entity

The sheet name can be changed in the block configuration, but since it is unlikely to be changed, a detailed explanation is omitted.

utterances sheet

Each row consists of the following columns

- `flag`

  Flags to be used or not. `Y` (yes), `T` (test), etc. are often written. Which flag's rows to use is specified in the configuration. In the configuration of the sample application, all rows are used.

- `type`

  User utterance type (Intent)

- `utterance`

  Example utterance. Each slot is annotated as `(<linguistic expression corresponding to the slot>)[<slot name>]`, as in `I like (chiken salad sandwiches)[favorite_sandwich].` Note that the linguistic expression corresponding to a slot does not always equal to the slot value that appears in the language understanding result (i.e., is sent to manager). If the linguistic expression eauals to the `synonyms` column of the `dictionary` sheet, the slot value will be the value of the `entity` column of the `dictionary` sheet.

The sheets that this block uses, including the utterance sheets, can have other columns than these.

slots sheet

Each row consists of the following columns.

- `flag`

  Same as on the utterance sheet.

- `slot name`

  Slot name. It is used in the example utterances in the utterances sheet. Also used in the language understanding results.

- `entity class`

  Entity class name. This indicates what type of noun phrase the slot value is. Different slots may have the same entity class. For example, `I want to buy an express ticket from (Tokyo)[source_station] to (Kyoto)[destination_station]`, both `source_station` and `destination_station` have entity of class `station`.

  You can use a dictionary function (of the form `dialbb/<function name>`) as the value of the `entity class` column. This allows you to obtain a dictionary description with a function call instead of writing the dictionary information on a dictionary sheet (e.g. `dialbb/location`).

  The function (e.g. `dialbb/location`) is described in "*Dictionary function definitions by developers*" below.

  The value of the entity class column can also be a Snips built-in entity. (e.g. `snips/city`)

  When you use Snips built-in entities, you need to install it as follows

  ```
  $ snips-nlu download-entity snips/city en
  ```

  Accuracy and other aspects of the Snips built-in entities have not been fully verified.

entities sheet

Each row consists of the following columns.

- `flag`

  Same as on the utterance sheet.

- `entity class`

  Entity class name. If a dictionary function is specified in the slots sheet, the same dictionary function name must be written here.

- `use synonyms`

  Whether to use synonyms or not. (`Yes` or `No`)

- `automatically extensible`

  Whether to recongize values not in dictionary or not. (`Yes` or `No`)

- `matching strictness`

  Strictness of matching entities. `0.0` - `1.0`

dictionary sheet

Each row consists of the following columns

- `flag`

  Same as that of the utterance sheet.

- `entity class`

  entity class name.

- `entity`

  The name of the dictionary entry. It is also included in language understanding results.

- `synonyms`

  Synonyms joined by `','`.

Dictionary function definitions by developers

Dictionary functions are mainly used to retrieve dictionary information from external databases.

Dictionary functions are defined in the module specified by `dictionary_function` in the block configuration.

The dictionary function takes configuration and block configuration as arguments. It is assumed that the these contain connection information to external databases.

The return value of a dictionary function is a list of dicts of the form `{"value": <string>, "synonyms": <list of strings>}`. The `synonyms"` key is optional.

Examples of dictionary functions are shown below.

```
def location(config: Dict[str, Any], block_config: Dict[str, Any]) \
    -> List[Dict[str, Union[str, List[str]]]]:
    return [{"value": "US", "synonyms": ["USA", "America"]},
            {"value": "California", "synonyms": ["CA"]},
            {"value": "Texas"}]
```

Snips training data

When the application is launched, the above knowledge is converted into Snips training data and a model is created.

The Snips training data is `_training_data.json` in the application directory. By looking at this file, you can check if the conversion is successful.

5.4 ChatGPT Understander (Language Understanding Block using ChatGPT)

(`dialbb.builtin_blocks.understanding_with_chatgpt.chatgpt_understander.Understander`)

Determines the user utterance type (also called intent) and extracts the slots using OpenAI's ChatGPT.

Performs language understanding in Japanese if the `language` element of the configuration is `ja`, and language understanding in English if it is `en`.

At startup, this block reads the knowledge for language understanding written in Excel, and converts it into the list of user utterance types, the list of slots, and the few shot examples to be embedded in the prompt.

At runtime, input utterance is added to the prompt to make ChatGPT perform language understanding.

5.4.1 Input/Output

- input
  - `input_text`: input string

    The input string is assumed to be canonicalized.

    * Example: `"I like chicken salad sandwiches"`.
- output
  - `nlu_result`: language understanding result (dict)

````
```json
 {
     "type": <user utterance type (intent)>,.
     "slots": {<slot name>: <slot value>, ... , <slot name>: <slot value>}
 }
```

The following is an example.

```json
 {
     "type": "tell-like-specific-sandwich",
     "slots": {"favorite-sandwich": "roast beef sandwich"}
 }
```
````

5.4.2 Block Configuration Parameters

- `knowledge_file` (string)

  Specifies the Excel file that describes the knowledge. The file path must be relative to the directory where the configuration file is located.

- `flags_to_use` (list of strings)

  Specifies the flags to be used. If one of these values is written in the `flag` column of each sheet, it is read. If this parameter is not set, all rows are read.

- `canonicalizer`

  Specifies the canonicalization information to be performed when converting language comprehension knowledge to Snips training data.

  - `class`

    Specifies the class of the normalization block. Basically, the same normalization block used in the application is specified.

- `knowledge_google_sheet` (hash)
  - This specfies information for using Google Sheet instead of Excel.
    - `sheet_id` (string)

      Google Sheet ID.

    - `key_file` (string)

      Specify the key file to access the Google Sheet API as a relative path from the configuration file directory.

- `gpt_model` (string. The default value is `gpt-3.5-turbo`.)

  Specifies the ChatGPT model. `gpt-4-turbo` can be specified. `gpt-4` cannot be used.

- `prompt_template`

  This specifies the prompt template file as a relative path from the configuration file directory.

  When this is not specified, `dialbb.builtin_blocks.understanding_with_chatgpt.prompt_templates_ja .PROMPT_TEMPLATE_JA` (for Japanese) or `dialbb.builtin_blocks.understanding_with_chatgpt.prompt_templates_en .PROMPT_TEMPLATE_EN` (for English) is used.

  A prompt template is a template of prompts for making ChatGPT language understanding, and it can contain the following variables starting with `@`.

  - `@types` The list of utterance types.
  - `@slot_definitions` The list of slot definitions.
  - `@examples` So-called few shot examples each of which has an utterances example, its utterance type, and its slots.
  - `@input` input utterance.

  Values are assigned to these variables at runtime.

5.4.3 Language Understanding Knowledge

Language understanding knowledge consists of the following two sheets.

sheet name	contents
utterances	examples of utterances by type
slots	relationship between slots and entities and a list of synonyms

The sheet name can be changed in the block configuration, but since it is unlikely to be changed, a detailed explanation is omitted.

utterances sheet

Each row consists of the following columns

- `flag`

  Flags to be used or not. `Y` (yes), `T` (test), etc. are often written. Which flag's rows to use is specified in the configuration. In the configuration of the sample application, all rows are used.

- `type`

  User utterance type (Intent)

- `utterance`

  Example utterance.

- `slots`

  Slots that are included in the utterance. They are written in the following form

  ```
  <slot name>=<slot value>, <slot name>=<slot value>, ... <slot name>=<slot value>
  ```

  The following is an example.

  ```
  location=philladelphia, favorite-sandwich=cheesesteak sandwitch
  ```

The sheets that this block uses, including the utterance sheets, can have other columns than these.

slots sheet

Each row consists of the following columns.

- `flag`

  Same as on the utterance sheet.

- `slot name`

  Slot name. It is used in the example utterances in the utterances sheet. Also used in the language understanding results.

- `entity`

  The name of the dictionary entry. It is also included in language understanding results.

- `synonyms`

  Synonyms joined by `','`.

5.5 STN Manager (State Transition Network-based Dialogue Management Block)

(`dialbb.builtin_blocks.stn_manager.stn_management`)

It perfomrs dialogue management using a state-transition neetwork.

- input
  - `sentence`: user utterance after canonicalization (string)

  - `nlu_result`: language understanding result (dictionary or list of dictionaries)
  - `user_id`: user ID (string)
  - `aux_data`: auxiliary data (dictionary) (not required, but specifying this is recommended)
- output
  - `output_text`: system utterance (string)

    Example:

    ```
    "So you like chiken salad sandwiches."
    ```

  - `final`: a flag indicating whether the dialog is finished or not. (bool)
  - `aux_data`: auxiliary data (dictionary type)

    The auxiliary data of the input is updated in action functions described below, including the ID of the transitioned state. Updates are not necessarily performed in action functions. The transitioned state is added in the following format.

    ```
    {"state": "I like a particular ramen" }
    ```

5.5.1 Block configuration parameters

- `knowledge_file` (string)

  Specifies an Excel file describing the scenario. It is a relative path from the directory wherer the configuration file exists.

- `function_definitions` (string)

  The name of the module that defines the scenario function (see *Dictionary function definitions by developers*). If there are multiple modules, connect them with `':'`. The module must be in the Python module search path. (The directory containing the configuration file is in the module search path.)

- `flags_to_use` (list of strings)

  Same as the Snips Understander.

- `knowledge_google_sheet` (object)

  Same as the Snips Understander.

- `scenario_graph`: (boolean. Default value is `False`)

  If this value is `True`, the values in the `system utterance` and `user utterance example` columns of the scenario sheet are used to create the graph. This allows the scenario writer to intuitively see the state transition network.

- `repeat_when_no_available_transitions` (Boolean. Default value is `False`)

  When this value is `True`, if there is no transition other than the default transition (see below) that matches the condition, the same utterance is repeated without transition.

5.5.2 Dialogue Management Knowledge Description

The dialog management knowledge (scenario) is written in the scenario sheet in the Excel file.

Each row of the sheet represents a transition. Each row consists of the following columns

- `flag`

  Same as on the utterances sheet.

- `state`

  The name of the source state of the transition.

- `system utterance`

  Candidates of the system utterance generated in the `state` state.

  The `{<variable>}` or `{<function call>}` in the system utterance string is replaced by the value assigned to the variable during the dialogue or the return value of the function call. This will be explained in detail in "*Variables and Function Calls in System Utterances*".

  There can be multiple lines with the same `state`, but all `system utterance` in the lines having the same `state` become system utterance candidates, and will be chosen randomely.

- `user utterance example`

  Example of user utterance. It is only written to understand the flow of the dialogue, and is not used by the system.

- `user utterance type`

  The user utterance type obtained by language understanding. It is used as a condition of the transition.

- `conditions`

  Condition (sequence of conditions). A function call that represents a condition for a transition. There can be more than one. If there are multiple conditions, they are concatenated with `';'`. Each condition has the form `<function name>(<argument 1>, <argument 2>, ..., <argument n>)`. The number of arguments can be zero. See *Function arguments* for the arguments that can be used in each condition.

- `actions`

  A sequece of actions, which are function calls to execute when the transition occurs. If there is more than one, they are concatenated with `;`. Each condition has the form `<function name>(<argument 1>, <argument 2>, ..., <argument n>)`. The number of arguments can be zero. See *Function arguments* for the arguments that can be used in each condition.

- `next state`

  The name of the destination state of the transition.

There can be other columns on this sheet (for use as notes).

If the `user utterance type` of the transition represented by each line is empty or matches the result of language understanding, and if the `conditions` are empty or all of them are satisfied, the condition for the transition is satisfied and the transition is made to the `next state` state. In this case, the action described in `actions` is executed.

Rows with the same `state` column (transitions with the same source state) are checked to see if they satisfy the transition conditions, **starting with the one written above**.

The default transition (a line with both `user utterance type` and `conditions` columns empty) must be at the bottom of the rows having the `state` column values.

5.5.3 Special status

The following state names are predefined.

- `#prep`

  Preparation state. If this state exists, a transition from this state is attempted when the dialogue begins (when the client first accesses). The system checks if all conditions in the conditions column of the row with the `#prep` value in the `state` column are met. If they are, the actions in that row's actions are executed, then the system transitions to the state in next state, and the system utterance for that state is outputted.

  This is used to change the initial system utterance and state according to the situation. The Japanese sample application changes the content of the greeting depending on the time of the day when the dialogue takes place.

  This state is not necessary.

- `#initial`

  Initial state. If there is no `#prep state`, the dialogue starts from this state when it begins (when the client first accesses). The system utterance for this state is placed in `output_text` and returned to the main process.

  There must be either `#prep` or `#initial` state.

- `#error`

  Moves to this state when an internal error occurs. Generates a system utterance and exits.

  A state ID beginning with `#final`, such as `#final_say_bye`, indicates a final state. In a final state, the system generates a system utterance and terminates the dialog.

5.5.4 Conditions and Actions

Contextual information

STN Manager maintains contextual information for each dialogue session. The contextual information is a set of variables and their values (python dictionary type data), and the values can be any data structure.

Condition and action functions access contextual information.

The context information is pre-set with the following key-value pairs.

key	value
_current_state_name	name of the state before transition (string)
_config	dictionary type data created by reading configuration file
_block_config	The part of the dialog management block in the configuration file (dictionary)
_aux_data	aux_data (dictionary) received from main process
_previous_system_utterance	previous system utterance (string)
_dialogue_history	Dialogue history (list)

The dialog history is in the following form.

```
[
  {
    "speaker": "user",
    "utterance": <canonicalized user utterance (string)>
  },
  {
```

(continues on next page)

(continued from previous page)

```
    "speaker": "system",
    "utterance": <canonicalized user utterance (string)
  },
  {
    "speaker": "user",
    "utterance": <canonicalized user utterance (string)
  },
  ...
]
```

In addition to these, new key/value pairs can be added within the action function.

Function arguments

The arguments of the functions used in conditions and actions are of the following types.

- Special variables (strings beginning with #)

  The following types are available

  - `#<slot name>`

    Slot value of the language understanding result of the previous user utterance (the input `nlu_result` value). If the slot value is empty, it is an empty string.

  - `#<key for auxiliary data>`

    The value of this key in the input `aux_data`. For example, in the case of `#emotion`, the value of `aux_data['emotion']`. If this key is missing, it is an empty string.

  - `#sentence`

    Immediate previous user utterance (canonicalized)

  - `#user_id`

    User ID string

- Variables (strings beginning with *)

  The value of a variable in contextual information. It is in the form `*<variable name>`. The value of a variable must be a string. If the variable is not in the context information, it is an empty string.

- Variable reference (string beginning with &)

  Refers to a contextual variable in function definitions. It is in the form `&<contextual variable name>`

- Constant (string enclosed in "")

  It means the string as it is.

5.5.5 Variables and Function Calls in System Utterances

In system utterances, parts enclosed in `{` and `}` are variables or function calls that are replaced by the value of the variable or the return value of the function call.

Variables that start with `#` are special variables mentioned above. Other variables are normal variables, which are supposed to be present in the context information. If these variables do not exist, the variable names are used as is without replacement.

For function calls, the functions can take arguments explained above as functions used for conditions or actions. The return value must be a string.

5.5.6 Function Definitions

Functions used in conditions and actions are either built-in to DialBB or defined by the developers.The function used in a condition returns a boolean value, while the function used in an action returns nothing.

Built-in functions

The built-in functions are as follows:

- Functions used in conditions
  - `_eq(x, y)`

    Returns `True` if `x` and `y` are the same.

    e.g., `_eq(*a, "b")` returns `True` if the value of variable `a` is `"b"`. `_eq(#food, "sandwich")`: returns `True` if `#food` slot value is `"sandwich"`.
  - `_ne(x, y)`

    Returns `True` if `x` and `y` are not the same.

    e.g., `_ne(#food, "ramen")` returns `False` if `#food` slot is `"ramen"`.
  - `_contains(x, y)`

    Returns `True` if `x` contains `y` as a string.

    e.g., `contains(#sentence, "yes")` : returns `True` if the user utterance contains "yes".
  - `_not_contains(x, y)`

    Returns `True` if `x` does not contain `y` as a string.

    e.g., `_not_contains(#sentence, "yes")` returns `True` if the user utterance contains `"yes"`.
  - `_member_of(x, y)`

    Returns `True` if the list formed by splitting `y` by `':'` contains the string `x`.

    e.g., `_member_of(#food, "ramen:fried rice:dumplings")`
  - `_not_member_of(x, y)`

    e.g., `_not_member_of(*favorite_food, "ramen:fried_han:dumpling")`
  - `_num_turns_exceeds(n)`

    Returns `True` when the number of user turns exceeds the integer represented by the string `n`.

    e.g.: `_num_turns_exceeds("10")`

  - `_check_with_llm(task)`

    Makes the judgment using a large language model. More details follow.

- Functions used in actions
  - `_set(x, y)`

    Sets `y` to the variable `x`.

    e.g., `_set(&a, b)`: sets the value of `b` to `a`.

    `_set(&a, "hello")`: sets `"hello"` to `a`.
  - `_set(x, y)`

    Sets `y` to the variable `x`.

    e.g., `_set(&a, b)`: sets the value of `b` to `a`.

    `_set(&a, "hello")`: sets `"hello"` to `a`.
- Functions used in system utterances
  - `_generate_with_llm(task)`

    Generates a string using a large language model (currently only OpenAI's ChatGPT). More details follow.

Built-in functions using large language models

The functions `_check_with_llm(task)` and `_generate_with_llm(task)` use a large language model (currently only OpenAI's ChatGPT) along with dialogue history to perform condition checks and text generation. Here are some examples:

- Example of a condition check:

```
_check_with_llm("Please determine if the user said the reason.")
```

- Example of text generation:

```
_generate_with_llm("Generate a sentence to say it's time to end the talk by␣
↪continuing the conversation in 50 words.")
```

To use these functions, the following settings are required:

- Set OpenAI's API key to environment variable `OPENAI_API_KEY`.

  Please check websites and other resources to find out how to obtain an API key from OpenAI.
- Add the following elements to the chatgpt block configuration:
  - `gpt_model` (string)

    This specifies the model name of GPT, such as `gpt-4-turbo`, `gpt-3.5-turbo`, etc. The default value is `gpt-3.5-turbo`. `gpt-4` cannot be used.
  - `temperature` (float)

    This specifies the temperature parameter for GPT. The default value is `0.7`.
  - `situation` (list of strings)

    A list that enumerates the scenarios to be written in the GPT prompt. If this element is absent, no specific situation is specified.

- persona (lis of strings)

  A list that enumerates the system persona to be written in the GPT prompt.

  If this element is absent, no specific persona is specified.

e.g.:

```
chatgpt:
  gpt_model: gpt-4-turbo
  temperature: 0.7
  situation:
    - You are a dialogue system and chatting with the user.
    - You met the user for the first time.
    - You and the user are similar in age.
    - You and the user talk in a friendly manner.
  persona:
    - Your name is Yui
    - 28 years old
    - Female
    - You like sweets
    - You don't drink alcohol
    - A web designer working for an IT company
    - Single
    - You talk very friendly
    - Diplomatic and cheerful
```

Syntax sugars for built-in functions

Syntax sugars are provided to simplify the description of built-in functions.

- <variable name>==<value>

  This means _eq(<variable name>, <value>).

  e.g.:

  ```
  #favorite_sandwich=="chiken salad sandwich"
  ```

- <variable name>!=<value>

  This means _ne(<variable name>, <value>).

  e.g.:

  ```
  #NE_Person!=""
  ```

- <variable name>=<value>

  This means _set(&<variable name>, <value>).

  e.g.:,

  ```
  user_name=#NE_Person
  ```

- $<task string>

  When used as a condition, it means _check_with_llm(<task string>), and when used in a system utterance enclosed in {}, it means _generate_with_llm(<task string>).

Example of a condition:

```
$"Please determine if the user said the reason."
```

Example of a text generation function call in a system utterance

```
I understand. {$"Generate a sentence to say it's time to end the talk by continuing␣
↪the conversation in 50 words" }  Thank you for your time.
```

Function definitions by the developers

When the developer defines functions, he/she edits a file specified in `function_definition` element in the block configuration.

```
def get_ramen_location(ramen: str, variable: str, context: Dict[str, Any]) -> None:
    location:str = ramen_map.get(ramen, "Japan")
    context[variable] = location
```

In addition to the arguments used in the scenario, variable of dictionary type must be added to receive contextual information.

All arguments used in the scenario must be strings. In the case of a special variable or variables, the value of the variable is passed as an argument. In the case of a variable reference, the variable name without the &' is passed, and in the case of a constant, the string in "" is passed.

5.5.7 Reaction

In an action function, setting a string to `_reaction` in the context information will prepend that string to the system's response after the state transition.

For example, if the action function `_set(&_reaction, "I agree.")` is executed and the system's response in the subsequent state is "How was the food?", then the system will return the response "I agree. How was the food?".

5.5.8 Continuous Transition

If a transition is made to a state where the first system utterance is `$skip`, the next transition is made immediately without returning a system response. This is used in cases where the second transition is selected based on the result of the action of the first transition.

5.5.9 Dealing with Multiple Language Understanding Results

If the input `nlu_result` is a list that contains multiple language understanding results, the process is as follows.

Starting from the top of the list, check whether the `type` value of a candidate language understanding result is equal to the `user utterance type` value of one of the possible transitions from the current state, and use the candidate language understanding result if there is an equal transition. If none of the candidate language comprehension results meet the above conditions, the first language comprehension result in the list is used.

5.5.10 Subdialogue

If the destination state name is of the form `#gosub:<state name1>:<state name2>`, it transitions to the state `<state name1>` and executes a subdialogue starting there. If the destination state is `:exit`, it moves to the state `<state name2>`. For example, if the destination state name is of the form `#gosub:request_confirmation:confirmed`, a subdialogue starting with `request_confirmatin` is executed, and when the destination state becomes `:exit`, it returns to `confirmed`. When the destination becomes `:exit`, it returns to `confirmed`. It is also possible to transition to a subdialogue within a subdialogue.

5.5.11 Advanced Mechanisms for Handling Speech Input

Additional block configuration parameters

- `input_confidence_threshold` (float; default value `1.0`) If the input is a speech recognition result and its confidence is less than this value, the confidence is considered low. The confidence of the input is the value of `confidence` in `aux_data`. If there is no `confidence` key in `aux_data`, the confidence is considered high. In the case of low confidence, the process depends on the value of the parameter described below.
- `confirmation_request` (object)

  This is specified in the following form.

```
confirmation_request:
  function_to_generate_utterance: <function name (string)>
  acknowledgement_utterance_type: <user utterance type name of acknowledgement␣
↪(string)>
  denial_utterance_type: <name of user utterance type for affirmation (string)>
```

  If this is specified, the function specified in `function_to_generate_utterance` is executed and the return value is spoken (called a confirmation request utterance), instead of making a state transition when the input is less certain. Then, the next process is performed in response to the user's utterance.

  - When the confidence level of the user's utterance is low, the transition is not made and the previous state of utterance is repeated.
  - If the type of user utterance is specified by `acknowledgement_utterance_type`, the transition is made according to the user utterance before the acknowledgement request utterance.
  - If the type of user utterance is specified by `denial_utterance_type`, no transition is made and the utterance in the original state is repeated.
  - If the user utterance type is other than that, a normal transition is performed.

  However, if the input is a barge-in utterance (`aux_data` has a `barge_in` element and its value is `True`), this process is not performed.

  The function specified by `function_to_generate_utterance` is defined in the module specified by `function_definitions` in the block configuration. The arguments of the function are the `nlu_result` and context information of the block's input. The return value is a string of the system utterance.

- `utterance_to_ask_repetition` (string)

  If it is specified, then when the input confidence is low, no state transition is made and the value of this element is taken as the system utterance. However, in the case of barge-in (`aux_data` has a `barge_in` element and its value is `True`), this process is not performed.

  `confirmation_request` and `utterance_to_ask_repetition` cannot be specified at the same time.

- ignore_out_of_context_barge_in (Boolean; default value is False).

  If this value is True, the input is a barge-in utterance (the value of barge_in in the aux_data of the request is True), the conditions for a transition other than the default transition are not met (i.e. the input is not expected in the scenario), or the confidence level of the input is low the transition is not made. In this case, the barge_in_ignored of the response aux_data is set to True.

- reaction_to_silence (object)

  It has an action element. The value of the action element is a string that can be either repeat or transition. If the value of the action element is "transition", the "destination" element is required. The value of the destination key is a string.

  If the input aux_data has a long_silence key and its value is True, and if the conditions for a transition other than the default transition are not met, then it behaves as follows, depending on this parameter:

  - If this parameter is not specified, normal state transitions are performed.
  - If the value of action is "repeat", the previous system utterance is repeated without state transition.
  - If the value of action is "transition", then the transition is made to the state specified by destination.

Adding built-in condition functions

The following built-in condition functions have been added

- _confidence_is_low()

  Returns True if the value of confidence in the input aux_data is less than or equal to the value of input_confidence_threshold in the configuration.

- _is_long_silence()

  Returns True if the value of long_silence in the input's aux_data is True.

Ignoring the last incorrect input

If the value of rewind in the input aux_data is True, a transition is made from the state before the last response. Any changes to the dialog context due to actions taken during the previous response will also be undone. This function is used when a user utterance is accidentally split in the middle during speech recognition and only the first half of the utterance is responded to.

Note that the contextual information is reverted, but not if you have changed the value of a global variable in an action function or the contents of an external database.

5.6 ChatGPT Dialogue (ChatGPT-based Dialogue Block)

(Changed in ver0.7)

(dialbb.builtin_blocks.chatgpt.chatgpt.ChatGPT)

Engages in dialogue using OpenAI's ChatGPT.

5.6.1 Input/Output

- Input
  - `user_utterance`: Input string (string)
  - `aux_data`: Auxiliary data (dictionary).
  - `user_id`: auxiliary data (dictionary)
- Output
  - `system_utterance`: Input string (string)
  - `aux_data`: auxiliary data (dictionary type)
  - `final`: boolean flag indicating whether the dialog is finished or not.

The inputs `aux_data` and `user_id` are not used. The output `aux_data` is the same as the input `aux_data` and `final` is always `False`.

When using these blocks, you need to set the OpenAI license key in the environment variable `OPENAI_API_KEY`.

5.6.2 Block Configuration Parameters

- `first_system_utterance` (string, default value is `""`)

  This is the first system utterance of the dialog.

- `user_name` (string, default value is `"User"`)

  This is used for the ChatGPT prompt. It is explained below.

- `system_name` (string, default value is `"System"`)

  This is used for the ChatGPT prompt. It is explained below.

- `prompt_template` (string)

  This specifies the prompt template file as a relative path from the configuration file directory.

  A prompt template is a template of prompts for making ChatGPT generate a system utterance, and it can contain the following variables starting with `@`.

  - `@dialogue_history` Dialogue history. This is replaced by a string in the following form:

```
<The value of system_name in the block configuration>: <system utterance>
<The value of user_name in the block configuration>: <user utterance>
<The value of system_name in the block configuration>: <system utterance>
<The value of user_name in the block configuration>: <user utterance>
...
<The value of system_name in the block configuration>: <system utterance>
<The value of user_name in the block configuration>: <user utterance>
```

- `gpt_model` (string, default value is `gpt-3.5-turbo`)

  Open AI GPT model. You can specify `gpt-4`, `gpt-4-turbo` and so on.

5.6.3 Process Details

- At the beginning of the dialog, the value of `first_system_utterance` in the block configuration is returned as system utterance.
- In the second and subsequent turns, the prompt template in which `@dialogue_history` is replace by the dialogue histor is given to ChatGPT and the returned string is returned as the system utterance.

5.7 spaCy-Based NER (Named Entity Recognizer Block using spaCy)

(`dialbb.builtin_blocks.ner_with_spacy.ne_recognizer.SpaCyNER`)

Performs named entity recognition using spaCy and GiNZA.

5.7.1 Input/Output

- Input
  - `input_text`: Input string (string)
  - `aux_data`: auxiliary data (dictionary)
- Output
  - `aux_data`: auxiliary data (dictionary)

    The inputted `aux_data` plus the named entity recognition results.

The result of named entity recognition is as follows.

```
{
  "NE_<label>": "<named entity>",
  "NE_<label>": "<named entity>",
  ...
}
```

<label> is a class of named entities. <named entity> is a found named entity, a substring of ``input_text. If multiple named entities of the same class are found, they are concatenated with ':`.

Example:

```
{
  "NE_Person": "John:Mary",
  "NE_Dish": "Chiken Marsala"
}
```

See the spaCy/GiNZA model website for more information on the class of named entities.

- `ja-ginza-electra` (5.1.2): https://pypi.org/project/ja-ginza-electra/
- `en_core_web_trf` (3.5.0): https://spacy.io/models/en#en_core_web_trf-labels

5.7.2 Block Configuration Parameters

- `model` (String: Required)

  The name of the spaCy/GiNZA model. It can be `ja_ginza_electra` (Japanese), `en_core_web_trf` (English), etc.

- `patterns` (object; Optional)

  Describes a rule-based named entity extraction pattern. The pattern is a YAML format of the one described in spaCy Pattern Description.

  The following is an example.

```
patterns:
  - label: Date
    pattern: yesterday
  - label: Date
    pattern: The day before yesterday
```

5.7.3 Process Details

Extracts the named entities in `input_text` using spaCy/GiNZA and returns the result in `aux_data`.