DialBB Document

Release v0.9.0

Dec 23, 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

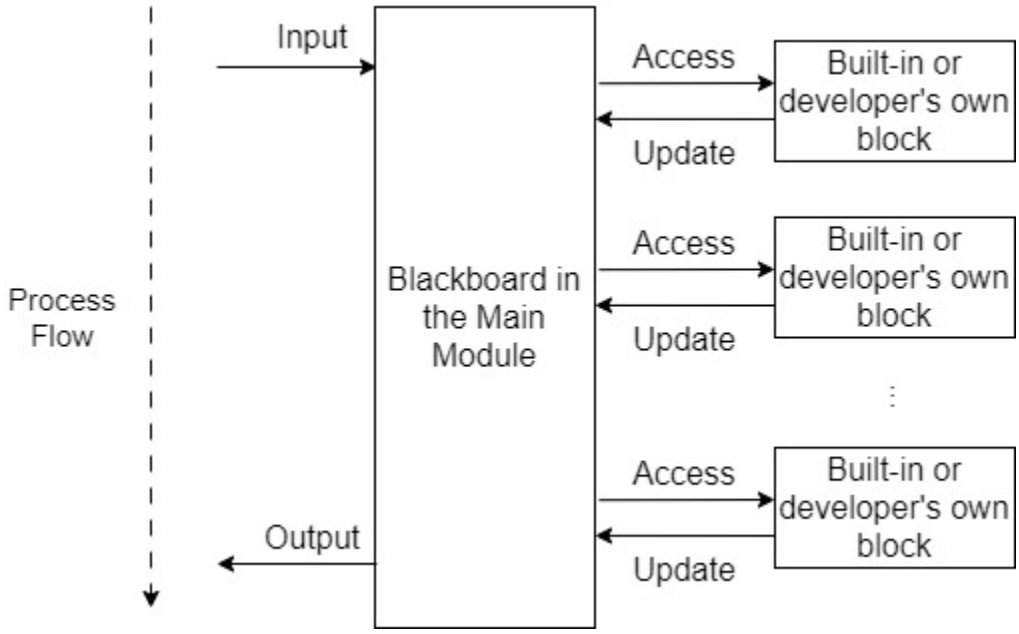
TWO

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

TUTORIAL

3.1 Introduction

DialBB comes with several sample applications. In this chapter, we will use the English applications among them to explain the structure of a DialBB application and the method for building an application using DialBB.

For instructions on how to run these applications, please refer to README.

3.2 Parrot Sample Application

3.2.1 Description

This is a simple application that just parrots back what the user says. It does not use any built-in block classes.

It can be found in `sample_apps/parrot`.

The configuration file that defines this application is located at `sample_apps/parrot/config.yml`, and its contents are as follows:

```
blocks:
  - name: parrot
    block_class: parrot.Parrot
    input:
      input_text: user_utterance
      input_aux_data: aux_data
    output:
      output_text: system_utterance
      output_aux_data: aux_data
      final: final
```

`blocks` element is a list of configurations for the blocks used in this application, referred to as *block configurations*. This application uses only one block.

`name` specifies the name of the block. This name is used in logs.

`block_class` specifies the class name of the block. An instance of this class is created to exchange information with the main module. The class name should be written as a relative path from the configuration file or from the `dialbb` directory.

The block class must be a subclass of `dialbb.abstract_block.AbstractBlock`.

`input` defines how information is received from the main module. For example:

```
input_text: user_utteranee
```

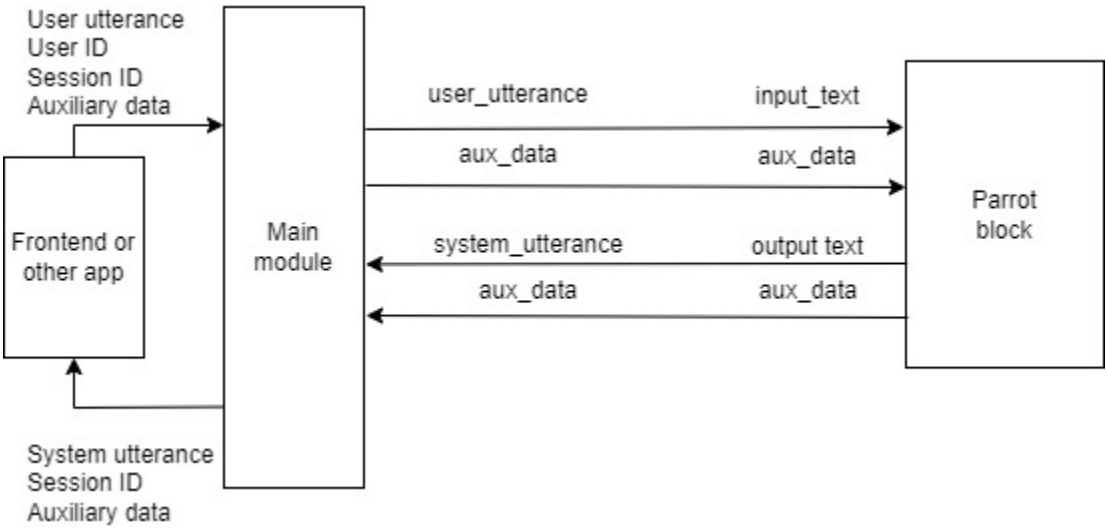
This means that the `blackboard['user_utterance']` in the main module can be referenced as the `input_text` element in the argument (dictionary type) of the `process` method of the block class.

`output` specifies the transmission of information to the main module. For example,

```
output_text: system_utterance
```

means that `blackboard['output_text']` in the main module is overwritten or appended with the `output_text` element in the output (dictionary type) of the `process` method of the block class.

When illustrated, it looks like the following:



The symbols above the arrows connecting the main module and the block represent the keys on each side: the left side shows the key in the `blackboard` of the main module, while the right side shows the key for input or output in the block.

Additionally, by looking at `sample_apps/parrot/parrot.py`, you should be able to better understand the concept of block classes in DialBB.

3.2.2 Debug Mode

By setting the environment variable `DIALBB_DEBUG` to `yes` as shown below, the log level will switch to debug mode.

```
export DIALBB_DEBUG=yes; python run_server.py sample_apps/parrot/config.yml
```

This will output detailed logs to the console, which should help deepen your understanding by observing the system's behavior in more detail.

3.3 ChatGPT Dialogue Application

3.3.1 Description

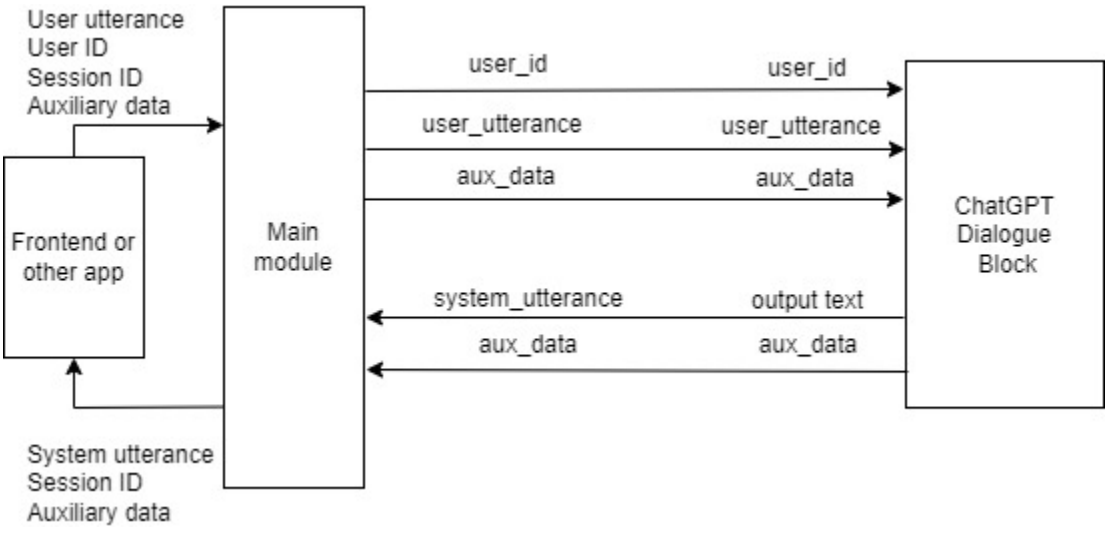
Using *ChatGPT Dialogue (ChatGPT-based Dialogue Block)*, we will conduct a dialogue with OpenAI's ChatGPT.

It can be found in `sample_apps/chatgpt/`.

The contents of `sample_apps/chatgpt/config_en.yml` are as follows.

```
blocks:
  - name: chatgpt
    block_class: dialbb.builtin_blocks.chatgpt.chatgpt.ChatGPT
    input:
      user_id: user_id
      user_utterance: user_utterance
      aux_data: aux_data
    output:
      system_utterance: system_utterance
      aux_data: aux_data
      final: final
    user_name: User
    system_name: System
    first_system_utterance: "Hello! Let's talk about food. What kind of cuisine do you␣
↪like?"
    prompt_template: prompt_template_en.txt
    gpt_model: gpt-4o-mini
```

The exchange of information with the main module is illustrated as follows.



In addition to `input` and `output`, several other parameters are configured as block configuration parameters.

The `prompt_template` specifies the template for the system prompt.

The contents of the prompt template `sample_apps/chatgpt/prompt_template_en.txt` are as follows.

```
# Task Description
```

(continues on next page)

(continued from previous page)

```
- You are a dialogue system and are chatting with the user on food. Please generate next␣
↪system utterance in less than 30 words.

# Your persona

- Emma
- female
- likes chocolates and wines
- working for an IT company
- very friendly and extrovert

#  Situation

- You first met the user.
- The user and you are in the same age group
- The user and you talk friendly

# The flow of dialogue

- Introduce each other
- Tell the user that you like Italian food
- Ask the user if se/he likes Italian food
- If the user likes Italian, ask the user which kind of Italian food she/he likes
- If the user doesn't like Italian, ask her/him why.

# Dialogue up to now

@dialogue_history
```

The dialogue history up to that point is inserted at the end of `@dialogue_history` in the following format.

```
System: Hello! Let's talk about food. What kind of cuisine do you like?
User: I like Italian food.
System: That's awesome! Italian food is so delicious. What's your favorite dish? Pasta,␣
↪pizza, or maybe something else?
User: I like pasta.
```

Here, "System" and "User" use the names specified in the configuration as `user_name` and `system_name`.

3.3.2 Creating an Application Using the ChatGPT Application

To create a new application by reusing this application, follow these steps:

- Copy the entire `sample_apps/chatgpt` directory. It doesn't need to be placed within the DialBB directory; any directory will work.
- Edit `config.yml` and `prompt_template_en.txt`. You can also rename these files if desired.
- Start the application with the following command:

```
export PYTHONPATH=<DialBB directory>; python run_server.py <configuration file>
```

3.4 Simple Application

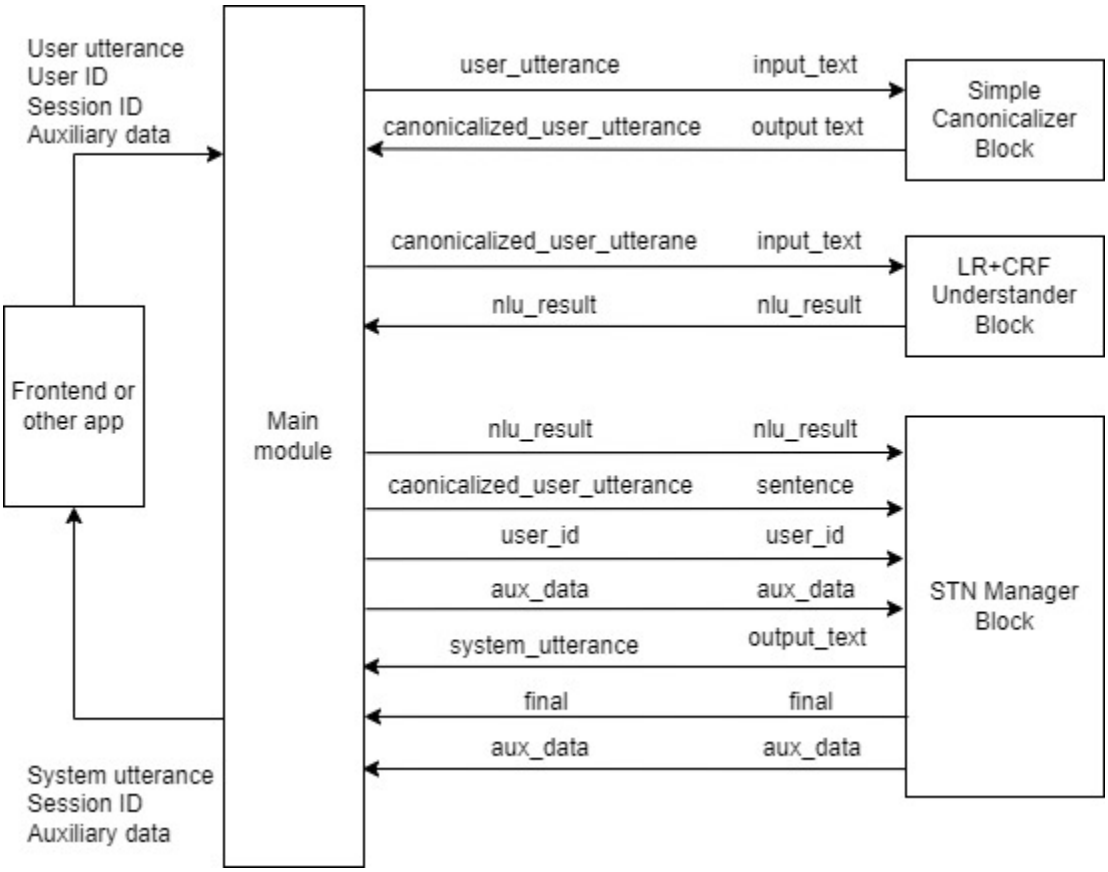
Here is a sample application using the embedded blocks.
(As of v0.9, it has been replaced with an application that does not use Snips NLU.)

- *Simple Canonicalizer (Simple String Canonicalizer Block)*
- *LR-CRF Understander (Language Understanding Block using Logistic Regression and Conditional Random Fields)*
- *STN Manager (State Transition Network-based Dialogue Management Block)*

It can be found in `sample_apps/simple_ja/`.

3.4.1 System Architecture

This application has the following system architecture.



This application uses the following three built-in blocks. Details on these built-in blocks are explained in Section 5.

- Simple Canonicalizer: Normalizes user input sentences (converting uppercase to lowercase, etc.).
- LR-CRF Understander Performs language understanding. It uses Logistic Regression and Conditional Random Fields (CRF) to determine user utterance types (also known as intents) and extract slots.
- STN Manager: Manages dialogue and generates language. Dialogue management is performed using a State Transition Network (STN), which outputs system responses.

3.4.2 Files Comprising the Application

The files that make up this application are located in the `sample_apps/simple_ja` directory (folder).

The `sample_apps/simple_ja` directory includes the following files:

- `config.yml`

  This is the configuration file that defines the application. It specifies which blocks to use and which files each block should load. The format of this file is explained in detail in Section 4.3.

- `config_gs_template.yml`

  This is a template configuration file for using Google Spreadsheet instead of Excel for the knowledge needed by the LR-CRF Understander block and the STN Manager block. You can use it by copying the file and adding information needed to access Google Spreadsheet.

- `simple-nlu-knowledge-ja.xlsx`

  This file contains the knowledge (language understanding knowledge) used by the LR-CRF Understander block.

- `simple-scenario-ja.xlsx`

  This file contains the knowledge (scenario) used by the STN Manager block.

- `scenario_functions.py`

  This is a program used by the STN Manager block.

- `test_inputs.txt`

  This is the test scenario used for system testing.

3.4.3 LR-CRF Understander Block

Language Understanding Result

The LR-CRF Understander block analyzes the input utterance and outputs the language understanding result.
The language understanding result consists of a type and a set of slots.

For example, the language understanding result for "I like roast beef sandwiches" would be as follows.

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

The phrase is categorized under the type `"tell-like-specific-sandwich"`, and the slot `"favorite-sandwich"` has the value `"roast beef sandwich"`. There may be utterances that contain multiple slots as well.

NLU Knowledge

The language understanding knowledge used by the LR-CRF Understander block is written in the `simple-nlu-knowledge-en.xlsx` file. For details on the description method of this language understanding knowledge, refer to `nlu_knowledge`. Below is a brief explanation.

The language understanding knowledge consists of the following two sheets:

Sheet Name	Content
utterances	Examples of utterances for each type and the slots that should be extracted from those utterances
slots	Relationship between slots and entities

Here is an excerpt from the *utterances* sheet.

flag	type	utterance	slots
Y	yes	yes	
Y	yes	yeah	
Y	tell-like-specific-sandwich	I like roast beef sandwiches	favorite-sandwich=roast beef sandwiches
Y	tell-like-specific-sandwich	chicken salad sandwiches are my favorite	favorite-sandwich=chicken salad sandwiches

The first row indicates that the type of "yes" is "yes", and it has no slots. The language understanding result for "yes" would be as follows:

```
{
   "type": "yes"
}
```

The language understanding result for "chicken salad sandwiches are my favorite" would be as follows:

```
{
  "type": "tell-like-specific-sandwich",
  "slots": {
     "favorite-sandwich": "chicken salad sadwiches"
  }
}
```

The `flag` column is used to specify in the configuration whether or not to use that row.

Next, here is a portion of the contents from the `slots` sheet.

flag	slot name	entity	synonyms
Y	favorite-sandwich	roast beef sandwich	roast beef, roast beef sandwiches
Y	favorite-sandwich	egg salad sandwich	egg salad sandwiches, Egg Salad

The `slot name` column indicates the slot name, `entity` specifies the slot value, and `synonyms` lists synonyms for that value.

For example, the first row indicates that if values like `roast beef` or `roast beef sandwiches` are obtained as slot values for the slot `favorite-sandwich` , they will be replaced with `roast beef sandwich` in the language understanding result.

Construction and Use of Language Understanding Models

When the application is launched, models based on logistic regression and conditional random fields (CRFs) are created using the aforementioned knowledge and are utilized during runtime.

3.4.4 STN Manager Block

Overview

The STN Manager block performs dialogue management and language generation using a State-Transition Network (STN). The State-Transition Network is also referred to as a "scenario." The scenario is documented on the `scenario` sheet within the `simple-scenario-ja.xlsx` file. For details on how to write this sheet, please refer to Section 5.4.2.

Scenario Description

A portion of the scenario description is provided below.

flag	state	system utterance	user utterance example	user utterance type	conditions	actions	next state
Y	like-sandwi	what kind of sandwich do you like?	I like egg salad sandwiches.	tell-like-specific-sandwich	_eq(#favorite-sandwich, "egg salad sandwich")	_set(&topic_sandw #favorite-sandwich)	egg-salad-sandwich
Y	like-sandwi		I like roast beef sandwiches.	tell-like-specific-sandwich	is_known_sandwich(sandwich)	_set(&topic_sandw sandwich)	known- e-sandwich
Y	like-sandwi		I like tomato and egg sandwich.	tell-like-specific-sandwich	is_novel_sandwich(# sandwich)		new-sandwich
Y	like-sandwi		Any sandwich is fine with me.				#final

Each row represents a single transition.

The `flag` column, similar to the language understanding knowledge, specifies whether that row will be used based on the configuration.

The `state` column contains the name of the initial state, and the `next state` column contains the name of the destination state.

The `system utterance` column contains the system's response output in that state. This response is linked to the value in the `state` column on the left, regardless of the transition in that row.

The `user utterance example` column provides an example of the expected user response for that transition. This example is not actually used in practice.

The `user utterance type` and `conditions` columns specify the conditions for the transition. The conditions for a transition are satisfied under the following conditions:

- The `user utterance type` column is empty, or the value in the `user utterance type` column matches the user utterance type of the language understanding result, and
- The `conditions` column is empty, or all conditions listed in the `conditions` column are satisfied.

These conditions are checked sequentially, starting from the topmost transition.

A row with both an empty `user utterance type` and an empty `conditions` column is referred to as a default transition. Generally, each state requires one default transition, which should be positioned at the bottom among the rows originating from that state.

Conditions

The `conditions` column contains a list of function calls representing specific conditions. If multiple function calls are present, they are separated by `;`.

The functions in the `conditions` column, known as *condition functions*, each return either `True` or `False`. When all function calls return `True`, the condition is considered satisfied.

Functions beginning with `_` are built-in functions. Other functions, defined in `scenario_functions.py` for this application, are custom functions created by developers.

The `_eq` function is a built-in function that returns `True` if two arguments contain the same string.

Arguments beginning with `#`, like `#favorite_sandwich`, are special arguments. For example, `#` followed by the name of a slot in the language understanding result denotes the value of the slot. `#favorite_sandwich` is the value of the `#favorite_sandwich` slot.

The values of the arguments enclosed by "" like `"tuna sandwich"` are the enclosed strings.

`_eq(#favorite_sandwich, "tuna sandwich")` returns `True` when the `favorite-sandwich` slot value is `tuna sandwich`.

The function `is_known_ramen(#favorite_sandwich)` is defined in `scenario_functions.py` so that it returns `True` when the system recognizes the value in the `favorite-sandwich` slot and returns `False` otherwise.

Condition functions can access data known as *context information* which is a dictionary-type data structure. Keys can be added to this structure within the condition or action functions. Some keys are pre-set with values, as detailed in the reference section `{numref}context_information`.

Actions

The `actions` column specifies the processes to be executed when a transition in that row occurs. It lists function calls; if there are multiple function calls, they are separated by `;`.

Functions used in the `actions` column are called action functions, and they do not return any values.

Similar to condition functions, functions that start with an underscore (_) are built-in functions. Other functions are custom functions created by the developer and are defined in `scenario_functions.py` for this application.

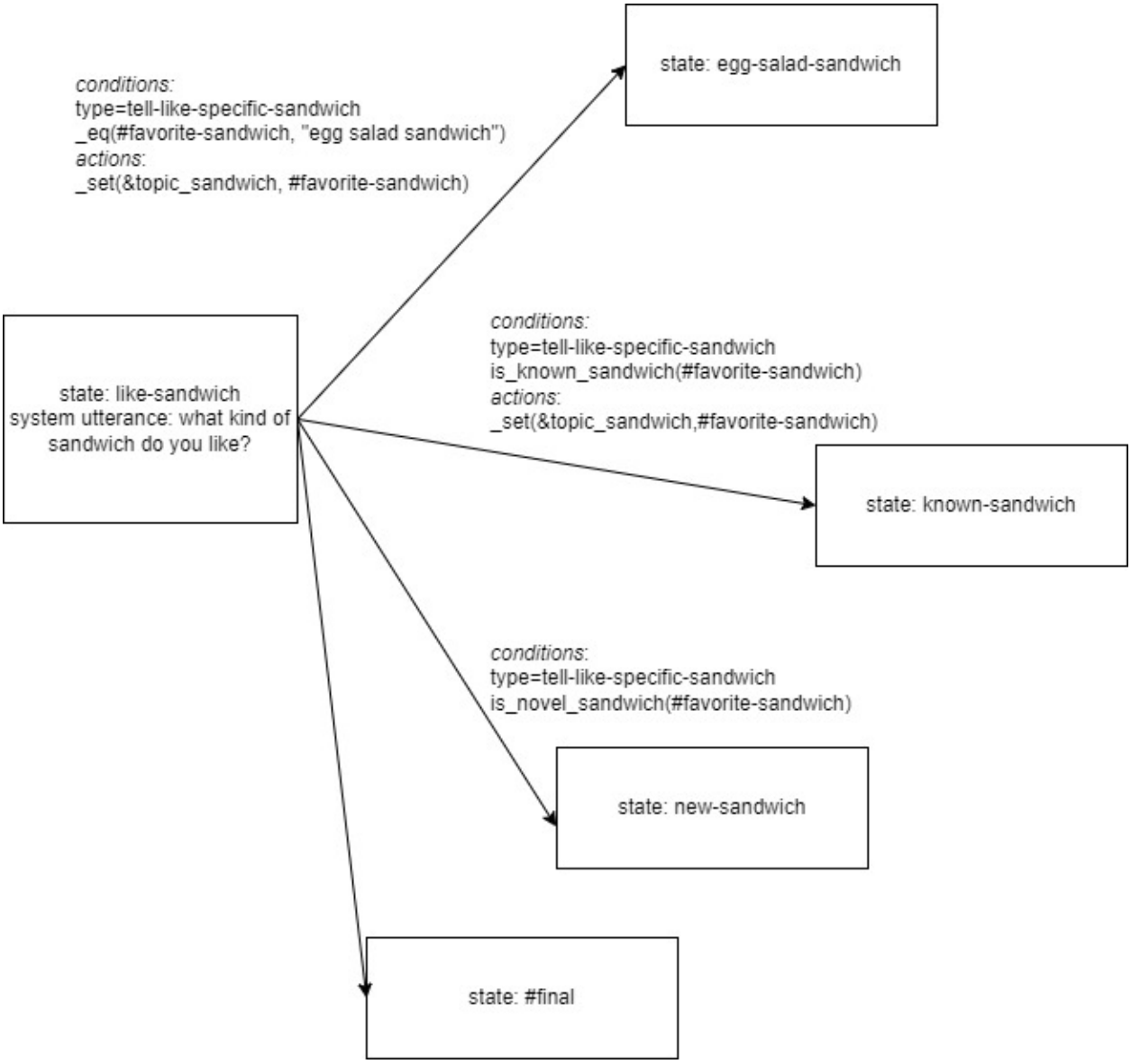
The `_set` function assigns the value of the second argument to the first argument. For instance, `_set(&topic_sandwich, #favorite-sandwich)` assigns the value of the `#favorite-sandwich` slot to the `topic_sandwich` key in the context information. Context information values can be accessed within conditions and actions by referencing `*<key name>`.

`is_known_sandwich(#favorite-sandwich)` is an example of a developer-defined function. This function, defined in `scenario_functions.py`, judges whether the value of the `favorite-sandwich` slot of the language understanding result is one of the sandwiches that the system knows using its known sandwich list.

Summary of Transition Description

To summarize, in the first row, when the state is `like-sandwich`, the system utters, "what kind of sandwich do you like?". If the type of the user's next utterance, as determined by language understanding, is `tell-like-specific-sandwich`, and the value of the `favorite-sandwich` slot is `egg salad sandwich` , then the conditions are met, and the transition occurs. The value of the `favorite-sandwich` slot, i.e., `egg salad sandwich`, is then set as the value for `topic_sandwich` in the context information, and the state changes to `egg-salad-sadwich`. If the conditions are not met, the conditions for the second row are checked.

A diagram would illustrate this as follows:



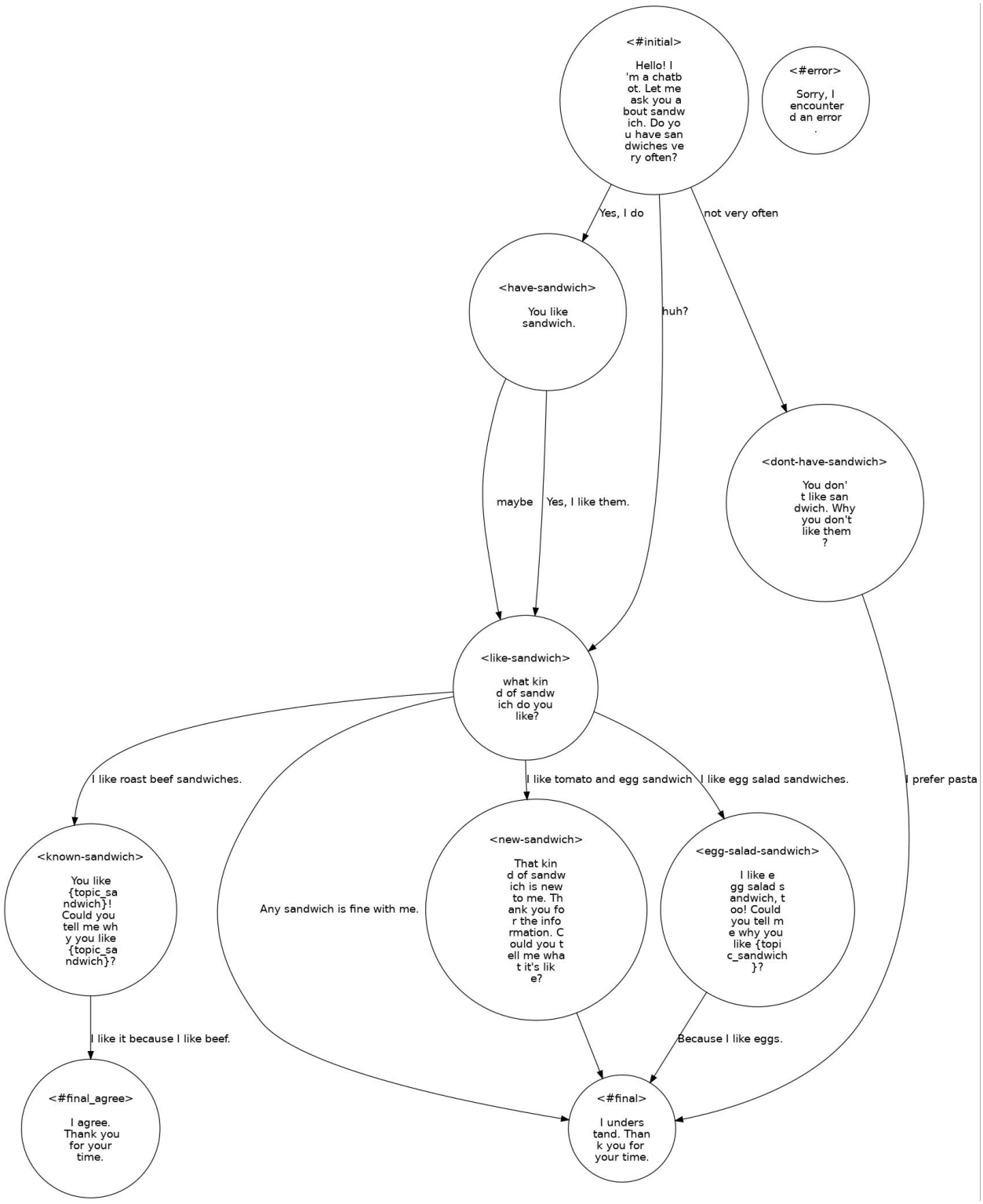
Special State Names

There are some special state names used in the system:

- `#prep`: This is the state before the dialogue begins. After the session starts, condition checks and actions are executed in this state.
- `#initial`: This state generates the first user prompt.
- States with names starting with `#final` are final states. They set the `final` value in the block's output to `True`, indicating the end of the dialogue.
- `#error`: This state is used when an internal error occurs. It also sets `final` to `True` in the block's output, signaling the end of the dialogue.

Scenario Graph

If Graphviz is installed, the application will output a graph (`_scenario_graph.jpg`) upon startup. This graph, or scenario graph, utilizes the system utterances from the `system utterance` column and the example user utterances from the `user utterance example` column to illustrate the dialogue flow. Below is the scenario graph for this application.


<#initial>
Hello! I 'm a chatb ot. Let me ask you a bout sandw ich. Do yo u have san dwiches ve ry often?
<#error>
Sorry, I encounter d an error .
Yes, I do
not very often
<have-sandwich>
You like sandwich.
huh?
<dont-have-sandwich>
You don' t like san dwich. Why you don't like them ?
maybe
Yes, I like them.
<like-sandwich>
what kin d of sandw ich do you like?
I like roast beef sandwiches.
I like tomato and egg sandwich
I like egg salad sandwiches.
I prefer pasta
<known-sandwich>
You like {topic_sa ndwich}! Could you tell me wh y you like {topic_sa ndwich}?
Any sandwich is fine with me.
<new-sandwich>
That kin d of sandw ich is new to me. Th ank you fo r the info rmation. C ould you t ell me wha t it's lik e?
<egg-salad-sandwich>
I like e gg salad s andwich, t oo! Could you tell m e why you like {topi c_sandwich }?
I like it because I like beef.
Because I like eggs.
<#final_agree>
I agree. Thank you for your time.
<#final>
I unders tand. Than k you for your time.

3.4.5 Utilizing N-Best Language Understanding Results

In this application, the LR-CRF language understanding block is set to output a 5-Best list of understanding results. This is specified by the `num_candidates` element in the configuration file.

```
blocks:  # bclock list
  - ....
  - name: understander
    ....
    num_candidates: 3
  - ....
```

In the STN Manager block, when checking the conditions for a transition, it examines the top language understanding results in order. If it finds a result that satisfies the conditions, it uses that result to execute the corresponding action and transitions to the next state.

3.4.6 Building an Application Using a Simple Application Structure

Overview

This section explains how to construct an application using a simple application structure, similar to the method in Section 3.3.2.

- **Copy the Application Directory**
  Copy the `sample_apps/simple_ja` directory as a whole. You can place it in any directory unrelated to the DialBB directory.
- **Edit Each File**
  You may edit each of the files in the directory as needed. Renaming the files is also permitted.
- **Run the Application**
  Start the application with the following command:

  ```
  export PYTHONPATH=<DialBB directory>;python run_server.py <configuration file>
  ```

Files to Modify

- `simple-nlu-knowledge-ja.xlsx`

  Edit this file to adjust the knowledge used in the LR-CRF language understanding block.

- `simple-scenario-ja.xlsx`

  Modify this file to update the scenario.

- `scenario_functions.py`

  Define the functions (conditional functions and action functions) that are added to the scenario in this file (see `custom_functions`).

  In each function definition, include an additional argument representing context information, typically defined as `context: Dict[str, Any]`. Context information encompasses both pre-registered data and data added during interaction by action functions. For further details, refer to Section 5.4.4.

- `config.yml`

  Modifications to this file are generally unnecessary if only basic functionality is being used.

3.5 Lab Application

3.5.1 Overview

It is based on language understanding by ChatGPT and network-based dialogue management, incorporating various functionalities of embedded blocks. The following embedded blocks are used:

- *Simple Canonicalizer (Simple String Canonicalizer Block)*
- *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)*

The main differences from simple applications are the use of the ChatGPT language understanding block and the spaCy named entity recognition block, as well as the advanced functionalities of the STN Manager block.

3.5.2 Files Comprising the Application

The files that make up this application are located in the `sample_apps/simple_ja` directory. This directory includes the following files:

- `config.yml`

  This is a configuration file that defines the application.

- `sample-knowledge-ja.xlsx`

  This file contains the knowledge used by the ChatGPT Language Understanding Block and the STN Manager Block.

- `scenario_functions.py`

  This is a program used within the STN Manager Block.

- `test_requests.json`

  This file contains test requests (refer to `test_requests`).

3.5.3 ChatGPT Language Understanding Block

Language understanding is conducted using ChatGPT in JSON mode. For this purpose, knowledge formatted in the same way as that used by the LR-CRF language understanding block is utilized. This knowledge is applied through few-shot learning to perform language understanding on the input text. The input and output of the LR-CRF language understanding block are also the same.

The specific GPT model for this process is defined in the block configuration under `gpt_model`.

The default prompt template is used for prompting ChatGPT to conduct language understanding. For further details, please refer to `chatgpt_understander_params`.

3.5.4 spaCy Named Entity Recognition (NER) Block

Named entity recognition (NER) is conducted using spaCy. The extracted entities are included in the `aux_data` section of the block's output. An example of the output format is as follows:

```
{"NE_PERSON": "John", "NE_DATE": "Friday"}
```

Here, `Person` and `Date` are entity classes defined within the spaCy model. These extracted entities can be accessed within the STN Manager block using special variables such as `#PERSON` and `#DATE`.

The specific model for spaCy is designated in the block configuration under `model`, with the current setting being `en_core_web_trf`.

Additionally, entity examples can be added to the `patterns` element within the block configuration to enhance extraction accuracy. The current configuration specifies examples as follows:

```
patterns:
  - label: Date
    pattern: yesterday
```

3.5.5 Functions of the STN Manager

In the experimental application, the following STN Manager functions, which are not used in the simple application, are utilized.

Function Calls and Special Variable References in System Utterances

In the simple application, only context information variables were embedded in system utterances. However, it is also possible to embed function calls and special variables.

For example:

```
I'm {get_system_name()}. If you don't mind, could you tell me your name?
```

In this case, the function `get_system_name(context)`, defined in `scenario_functions.py`, is called, and its return value (a string) replaces `{get_system_name()}`. This function is designed to return the value of `system_name` specified in the configuration file.

Another example:

```
Thank you {#NE_PERSON}! Let me ask you about sandwich. Do you have sandwiches very often?
```

Here, `{#NE_PERSON}` is replaced by the value of the special variable `#NE_PERSON`. `#NE_PERSON` corresponds to the value of `NE_PERSON` in `aux_data`, meaning it holds the value of the "Person" entity extracted by the named entity recognition block.

Syntax Sugar

Syntax sugar is provided to simplify the notation for calling built-in functions within the scenario. For instance, `confirmation_request="Sorry to ask you again, but do you often eat sadwiches?"` is equivalent to writing `_set(&confirmation_request, "Sorry to ask you again, but do you often eat sadwiches?")`.

Similarly:

- `#favorite-sandwich=="egg salad sandwich"` is equivalent to `_eq(#favorite-sandwich, "egg salad sandwich")`
- `#NE_PERSON!=""` is equivalent to `_ne(#NE_PERSON, "")`

This shorthand makes the scenario scripting more concise and easier to read.

Reaction Utterance Generation

In the `actions` section of the scenario, there is `_reaction="Great!"`. `_reaction` is a special variable in the context information that appends its value to the beginning of the next system utterance. For example, if the scenario transitions to a state called `like-sandwich` after this, the system utterance `what kind of sandwich do you like?` will have `Great!` added at the start, resulting in `Great! what kind of sandwich do you like?`.

By including such reactions, the system can acknowledge the user's previous input, enhancing the user experience by conveying that it is actively listening to what the user is saying.

Utterance Generation and Condition Evaluation Using ChatGPT

In a system utterance, the notation `$"Generate a sentence to say it's time to end the talk by continuing the conversation in 50 words"` is syntax sugar for the built-in function call `_check_with_llm("Generate a sentence to say it's time to end the talk by continuing the conversation in 50 words")`, which uses ChatGPT to generate an utterance.

Similarly, in the `conditions` section, the notation `$"Please determine if the user said the reason"` is syntax sugar for the built-in function call `_check_with_llm("Please determine if the user said the reason")`. This uses ChatGPT to evaluate the conversation history and returns a Boolean value indicating whether the user provided a reason.

The specific ChatGPT model, temperature parameter, context setting, and persona for utterance generation are specified in the block configuration as follows:

```
chatgpt:
  gpt_model: gpt-4o-mini
  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
```

(continues on next page)

(continued from previous page)

```
  - A web designer working for an IT company
  - Single
  - You talk very friendly
  - Diplomatic and cheerful
```

Sub-dialogue

The `next state` field contains `#gosub:confirmation-requested:confirmed-if-have-sandwich`, which indicates that after transitioning to the `confirmation_requested`state and conducting a dialogue, the conversation will return to the `confirmed-if-have-sandwich`state. A conversation starting from the `confirmation_requested` state is called a sub-dialogue. When the sub-dialogue transitions to the `#exit` state, it exits the sub-dialogue and goes to the ``confirmed-if-have-sandwich` state.

By setting up reusable dialogues as sub-dialogues for various situations that require user confirmation, you can reduce the amount of scenario writing required.

Skip Transition

When `$skip` is placed in the system utterance field, no system utterance is returned, and the system immediately moves to the next transition after evaluating conditions. This is useful when you want to determine the next transition based on the outcome of an action.

Repeat Function

The `repeat_when_no_available_transitions` option is specified in the block configuration. When this is enabled, if there are no transitions that satisfy the conditions, it returns to the original state and repeats the same utterance. In this case, it's acceptable to have states without default transitions. In this application, the `like-sandwich` state lacks a default transition, so if an unrelated utterance is made in this state, the same system utterance will be repeated.

Handling Voice Input

The STN Manager includes functionality to handle voice input, which can be utilized by configuring it in the block configuration. In this application, it is set up as follows:

```
input_confidence_threshold: 0.5
confirmation_request:
  function_to_generate_utterance: generate_confirmation_request
  acknowledgement_utterance_type: "yes"
  denial_utterance_type: "no"
#utterance_to_ask_repetition: "could you say that again?"
ignore_out_of_context_barge_in: yes
reaction_to_silence:
  action: repeat
```

Please refer to `handling_speech_input` for the meanings.

The file `test_requests.json` contains examples of inputs that support speech input.

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 LR-CRF Understander (Language Understanding Block using Logistic Regression and Conditional Random Fields)

(`dialbb.builtin_blocks.understanding_with_lr_crf.lr_crf_understander.Understander`)

Determines the user utterance type (also called intent) and extracts the slots using logistic regression and conditional random fields.

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 trains the models for logistic regression and conditional random fields.

At runtime, it uses the trained models for language understanding.

5.2.1 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>}},␣
↪...
 ....]
```

5.2.2 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.

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

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

- `knowledge_google_sheet` (hash)
  - This specifies information for using Google Sheets 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.

5.2.3 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 `','`.

5.3 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.

5.3.1 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"}
 }
```
````

5.3.2 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.

- `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.

5.3.3 Language Understanding Knowledge

The description format of the language understanding knowledge in this block is exactly the same as that of the LR-CRF Understander. For more details, please refer to "*Language Understanding Knowledge*" in the explanation of LR-CRF Understander.

5.4 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" }
  ```

5.4.1 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). 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 that matches the condition, the same utterance is repeated without transition.

5.4.2 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.

Unless `repeat_when_no_available_transitions` is `True`, the default transition is necessary.

5.4.3 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.

5.4.4 Conditions and Actions

Context information

STN Manager maintains context information for each dialogue session. The context 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 context 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)>
  },
  {
    "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 context 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 context variable in function definitions. It is in the form `&<context variable name>`

- Constant (string enclosed in "")

  It means the string as it is.

5.4.5 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.

5.4.6 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
```

(continues on next page)

(continued from previous page)

```
    - 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 context 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.

5.4.7 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?".

5.4.8 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.

5.4.9 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.

5.4.10 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.

5.4.11 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 context 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.

5.5 ChatGPT Dialogue (ChatGPT-based Dialogue Block)

(Changed in ver0.7)

(`dialbb.builtin_blocks.chatgpt.chatgpt.ChatGPT`)

Engages in dialogue using OpenAI's ChatGPT.

5.5.1 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`.

5.5.2 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.

5.5.3 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.

5.6 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.

5.6.1 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

5.6.2 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
```

5.6.3 Process Details

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

CHAPTER

SIX

APPENDIX

6.1 Discontinued Features

6.1.1 Snips Understander Built-in Block

As Snips has become challenging to install with Python 3.9 and above, it was discontinued in version 0.9. Please use the LR-CRF Understander built-in block as an alternative.

6.1.2 Whitespace Tokenizer and Sudachi Tokenizer Built-in Blocks

These blocks were discontinued in version 0.9. If you use LR-CRF Understander or ChatGPT Understander, there is no need for the Tokenizer blocks.

6.1.3 Snips+STN Sample Application

This sample application was discontinued in version 0.9.