# MACA: A Modular Architecture for Conversational Agents

Hoai Phuoc Truong[*], Prasanna Parthasarathi[†], and Joelle Pineau[‡]
School of Computer Science
McGill University

May 4, 2017

* phuoc.truong2@mail.mcgill.ca
† prasanna.p@cs.mcgill.ca
‡ jpineau@cs.mcgill.ca

## Contents

# 1 Introduction

## 1.1 Motivation

Growing interest in the research of conversational agents has propelled our ways to sophisticated approaches. As there is a potential commercial usecase and a lot of soup for researchers to ponder with the conversational agents, it is useful to have a open framework providing a unified view for a dialogue system that is derived from the fundamental units of the system and tested for its applicability across the more advanced models. To make the development of dialog architectures open and accessible and enable swifter development of dialog systems, we propose a unified architecture to be used to design, build and test a dialog agent from scratch.

# 2 System overview

## 2.1 High Level Architecture

An illustrative layout of the domain independent architecture for dialog agents is shown in Figure 1. The architecture pipeline consists of five major units – *Input, Pre-processing, Dialog Model, Post- processing, Output and Listeners*. Each unit has independent components that interact across the units. The architecture is designed in an object oriented fashion that abstracts the underlying implementations and thereby allow extension to be straightforward. This helps in block-wise designing of newer architectures preserving the original functionality yet providing a free hand in customizing each of them. A brief description of the system is presented in the figure below:

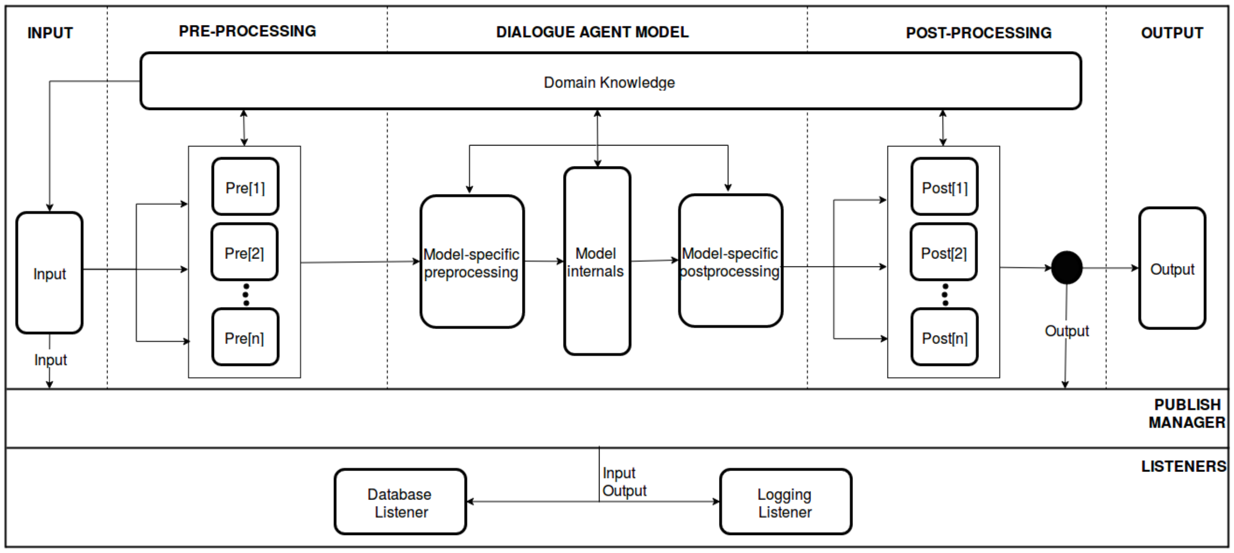

Figure 1: A unified architecture for dialog agents.

## 2.2 Domain knowledge

Domain knowledge contains static background information about the subject of the conversation. These are training data (e.g. conversations), constants, jargons, or context of the dialogue that the model is trying to generate. Data stored in domain knowledge must be independent of the model implementation, and should be sharable between different models.

## 2.3 Input

Input unit in the pipeline provides options from different input sources like *command line, web (mTurk), database etc.*. These different input classes help in tuning the architecture to different operating modes – *Data collection, Training*. The architecture renders a generic layout to be used for training a dialogue agent as well as testing it against another dialog or human agent for collecting data.

## 2.4 Preprocessing

The pre-processing unit has independent objects of several pre-processing classes including but not limited to:

- slot-filling
- word level parsing
- sentence parsing
- keyword tagging
- POS tagging
- sentence2vec
- word2vec

Objects within the pre-processing unit can interact with the domain knowledge module to parse domain specific information that may provide better detailing of the input.

The system architect may choose one or several of the pre-processing objects to include in this unit. These pre-processing will be performed in parallel and their results will be fed into the next unit as an array ordered respectively. This allows the dialog model to have more than one input representation. For cases in which a particular sequential pre-processing pipeline is preferred, the architect can adapt to this system by enabling a flag in the configuration file.

## 2.5 Dialogue model

Dialog Model unit has a specific layout for a dialog model that is broken down into three sub-components:

1. Model Specific Pre-processing
2. Model Internals
3. Model Specific Post-processing.

Model specific pre-processing and post-processing components are provided to give the luxury of designing fine-tuned pre-processing for a model. Model specific pre-processing component is necessary for models to transform the pre-processed input(s) into appropriate object representation that the model supports (e.g. matrix, lookup table, etc). On the other hand, model specific post-preprocessing component is necessary for models to transform their outputs to be comprehensible by the next independent layer in the system (e.g. matrix/vector representation to array of words/sentences).

In addition, as the model, in a generic sense, may also be an ensemble of dialog models, the model specific pre and post processing components can also be used to keep processing units that are specific to each of the model in the ensemble. For example, in an ensemble of models the model specific pre-processing unit may be used to separate and feed the inputs parsed from the pre-processing block to corresponding models.

*Model internals* comprises the dialogue architecture, which may be an existing model like HRED, Dual Encoder, POMDP etc or a newly designed model. This component takes its input(s) from model specific pre-processing unit and may do retrieval based or generative approach. The necessary vocabulary or responses as in the case of HRED and Dual Encoder, or the matrices as in the case of POMDP based approaches is stored in domain knowledge and is readily accessible. The dialog internals and model-specific pre/post-processing mutually share the model information and similar to the pre-processing unit, they can communicate with the domain knowledge unit. Development of abstract model internals corresponding to different classes of models could be considered in the future stages of the project.

## 2.6 Postprocessing

This block provides choices to choose the response, in the case of multi-response retrieval or response alteration based on linguistic characteristics, or to modify a response in accordance with the domain. It may as well be a translation of text to system call, which will encompass the POMDP case. Similar to the pre-processing unit, objects in post-processing unit will process the output in parallel and the system architect may choose to construct a sequential pipeline if need be. In addition, these post-processing objects within the post-processing unit can query the domain knowledge to obtain relevant data required for generation of text response.

## 2.7 Output

The architecture provides a generic way to output the response to the appropriate audience depending on the mode of use. Several implemented options are *command line, web based (mTurk) or data-base*. Logging of conversation in data collection mode (see Data Collection Mode section) can be done using a special *Output* module that writes to database.

## 2.8 Peripherals

### 2.8.1 Pubsub system

In addition to the main pipeline presented above, the prosposed system also includes a pubsub system aims to facilitate monitoring and independent evaluation of the model. This pubsub system allows the user to choose or plug in a wide range of peripheral modules to passively monitor the main system for execution behavior and performance. In addition to several default channels (see Operation modes section below) that the system will write to and read from, users can freely add their own channels to communicate between the main modules and the peripherals modules.

### 2.8.2 Scoring/Evaluation Mechanism

The scoring/evaluation module is a passive listener whose goal is to independently evaluate the performance of the current system. The implementation of this module therefore should not be dependent on any of the component of the main modules in the system to truly provide an objective evaluation of the system's performance. This module will have input-output pairs of the system as its data for evaluation based on various different evaluation criteria.

### 2.8.3 Database

The database is also a passive listener whose goal is to listen to any other channel of the running system and perform logging of relevant information for the current conversation. The module can be used to log the performance metrics and operational information of the system. Depending on the underlying database engine below, this module will have appropriate according implementation to organize the logged conversation.

## 3 Implementation Highlights

MACA's current implementation is in Python and includes standard libraries to ensure the framework's portability, as well as to facilitate rapid prototyping of different dialogue model strategies. Each component of the framework (e.g. *Input* component) is described with an abstract Python class, whose concrete implementation instances (i.e. Python objects) are manifestations of that component (e.g. Command line input, Database input). This corresponds to the abstraction layer of the architecture's module to foster independence of the pipeline implementation from that of the underlying dialogue model(s). The assembly of these components are then specified in a central configuration file representing an instantiation of the architecture. With this design, changes in the instantiation specifications can be done within the central configuration file by modifying the names of invoked modules. On the other hand, this setup allows system specifications to be completely contained within the central configuration file, which reduces maintenance effort and simplifies configuration modification during development. In addition, the open source nature of the framework encourages sharing and reusing of components, which allows researchers to easily develop from existing models and save time by reusing common components written by others.

## 4 Specific systems

This section presents the architectural details of the special systems adapted to the existing framework.

### 4.1 Goal oriented model

The interested goal oriented model involves a set of intentions for the users to choose from. Each intention consists of multiple information slots to be fulfilled. Once the dialog agent successfully obtained all information in the information slots, it confirms the collected information with the user. Upon user confirmation, the agent performs a terminal action based on the provided information, and considers the operation finished. The following describes an abstract goal oriented conversation in the discussed context:

### 4.2 Mturk data collection

Here we implemented a sample Mturk integration scheme that utilizes different components of this framework for data collection purpose. We aimed to demonstrate two important mturk experiments: response collection and model scoring. A typical mturk experiment is setup with an mturk server that serves mturk clients and record their results. However, we opted to implement our own server running the dialog framework in order to have more control of the data collection process. This allows us to have the results directly recorded in our server and avoid using the conventionally collected in mturk server.

#### 4.2.1 Response collection

The goal of this mturk experiment is to collect user responses to different contexts. This is particularly useful when generating a complete data set from a set of previously mined context. The following is the flow in a user response collection session with mturk.

1. Client requests a new context from the server.
2. Server responds to client request with a context.
3. Client receives the context and proceeds to display it to the user.
4. User inputs a response to the fetched context and clicks the submit button.
5. Client sends user's response to the server and request a new context at the samme time.

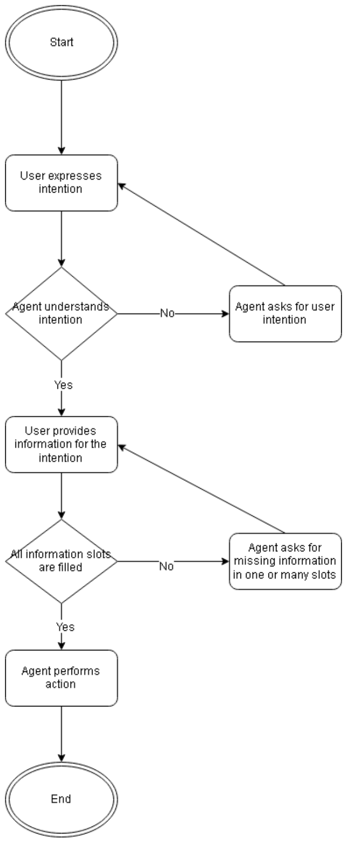
Start
User expresses intention
Agent understands intention
No
Agent asks for user intention
Yes
User provides information for the intention
All information slots are filled
No
Agent asks for missing information in one or many slots
Yes
Agent performs action
End

Figure 2: Flowchart for an abstract goal oriented conversation dialogue.

6. Go back to step 2 unless a sufficient amount of responses has been collected, or the user decides to stop.

To record the responses at the back end, we feed the provided contexts as input and user's responses as output to an instance of the dialog system with a void model. A database listener attached to the running system is responsible for writing down to files the conversation.

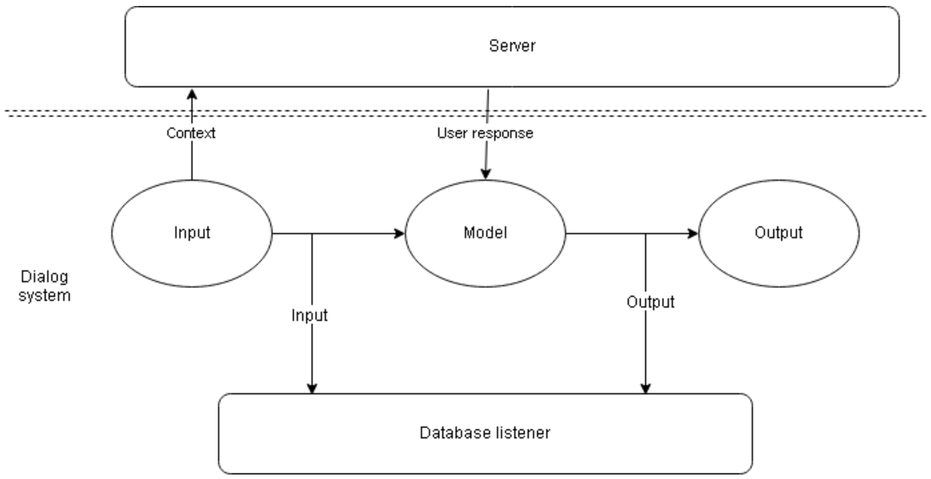

Figure 3: Mturk response collection at the back end.

Since there can be multiple users providing the responses to different contexts at once, asynchronous collection is preferred. This means that the server dialog system proceeds to response nothing upon receiving the context from the server to ensures that the pipeline is cleared in case of another context being inputted. Once a user response is received, it will immediately be outputted. We include a unique client id and a unique context-response id for each object being passed in the system so that the database listener can listen to these asynchronous messages and arrange them properly into meaningful conversations. To facilitate the additional id field without compromising existing implementations of the models, we introduce a simple wrapper around each processing unit in the pipeline (i.e. preprocessing, model, postprocessing).

### 4.2.2 Model scoring

The goal of this mturk experiment is to have a trained dialog system converse with users, and ask users to score each response from the system with an integer score from 1 to 5. The following is the flow in a user scoring session with mturk.

1. User inputs a new context.
2. Client sends the user's context to the server and request a response.
3. Server asks the trained dialog system for a response and respond to the client.
4. User receives the response, scores it and also provides a new context.
5. Client sends the provided scoring and new context to the server, asking for a response.
6. Go back to step 3 unless a sufficient amount of scoring has been collected, or the user decides to stop.

To record the responses at the back end, we feed the provided user context as input to an instance of the dialog system hosting the trained model. A database listener attached to the running system is responsible for writing down to files the conversation. The scoring provided user will also be published on the scoring channel for the database listener to record.

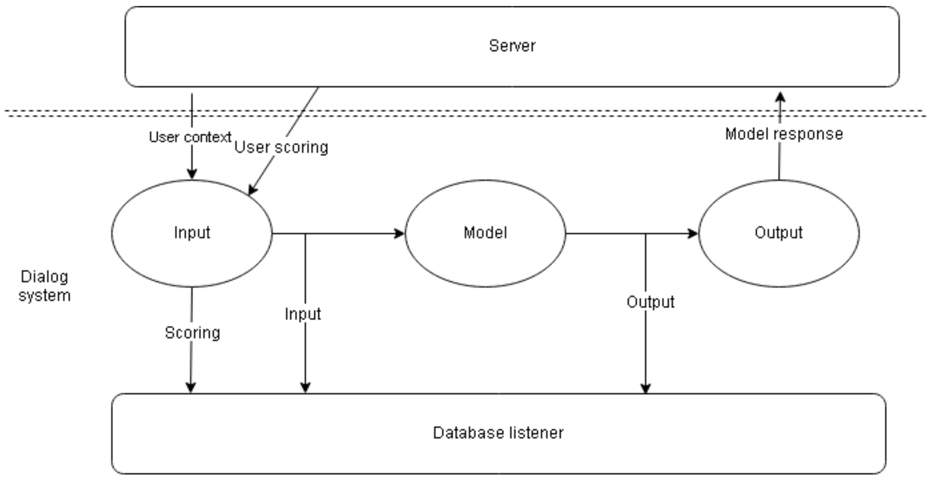

Figure 4: Mturk scoring collection at the back end.

As opposed to the asynchronous nature of the response collection experiment, this experiment requires input to be passed to the model the output to be fetched from the model before the server can respond to the client. We queue the received context at the input device and feed them to the model one by one to receive the according responses.

## 5 Operation modes

MACA can be operated in three different modes: *Data Collection*, *Training* and *Execution*. This section describes the data flow in the architecture along with abstract setups of the framework's components in these different operation modes for several dialogue models from the recent literature.

## 5.1 Data Collection Mode

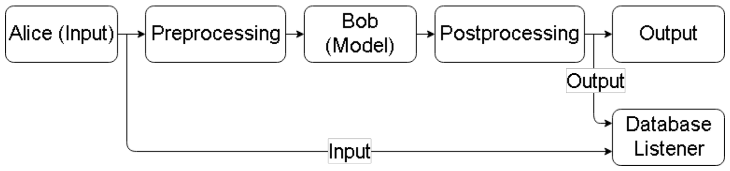

Figure 5: Data flow in data collection mode.

The goal of the data collection mode is to collect conversations as training datasets for dialogue models. In this mode, the two agents *Alice* and *Bob* involved in the conversation are considered the *Input* component and the *Dialogue Model* component respectively. Figure 5 describes a typical setup for the data collection process with said configuration. The conversation is recorded using a database listener that receives both input (context) and output (response) for each speaking turn, similar to the scheme presented in section 5.3 above.

This setup realizes the infrastructure required for two common dialogue data collection scenarios. The first scenario is collection of both contexts and responses. In this case, both agents are humans. In the second scenario, the goal is to collect human responses for a given set of contexts. In this case, agent Alice can be an implementation of the *Input* component fetching contexts from a database, while Bob is a human agent responding to the fetched contexts.

## 5.2 Training Mode

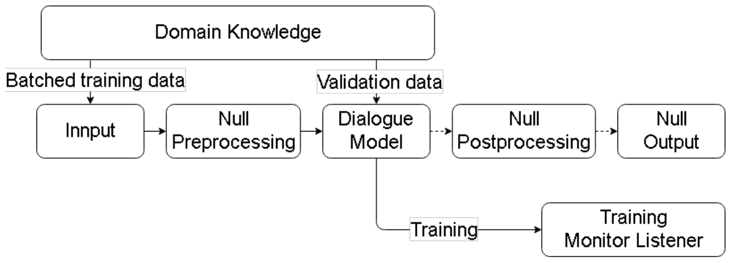

Figure 6: Data flow in training mode.

The goal of the training and validation mode is to use the data obtained in the data collection stage to train one or multiple dialogue models, as illustrated in figure 6. Assuming a dataset is available from the *Domain Knowledge* component, training data can be fetched as batches by the *Input* component and fed into the *VoidPreprocessing* component. This component simply forwards the data as is to the *Dialogue Model* component, which performs model training, and occasionally queries the domain knowledge for validation data to verify its training progress. Since system output is irrelevant within the training scenario, *Post-processing* and *Output* components are implemented with null operations, which simply discard their received contents. Once certain validation accuracy is achieved, the model can save its internals on to the disk and terminate the system. In addition to the core training process, the architect may opt to emit training information to a listener through the *training* channel to monitor the training progress.

## 5.3 Execution Mode

Data flow in execution mode is illustrated in figure 7. In this mode, all core components in the system are enabled and active. Given that the dialogue model has been successfully trained and fine-tuned, its internal states (e.g. weights, hyper-parameters) are loaded into the *Dialogue Model* component at system initialization time. Input data is retrieved in real time (through local user interface (e.g. terminal, GUI) or via an interface with the Internet

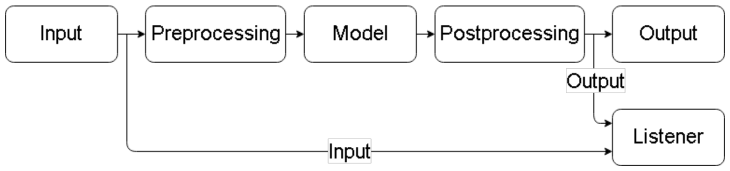

Figure 7: Data flow in execution mode.

(e.g. web page, chat client)). This input then enters the pipeline and goes through *Preprocessing*, *Dialogue model*, *Postprocessing* and finally *Output* component. At the end of the pipeline, the output component is responsible for sending the generated responses to relevant audiences (e.g. print to stdout, HTTP response, ...).

From the peripheral components perspective, conversation logging and system monitoring can be done through two default channels: *input* and *output*. Specifically, as shown in figure 7, the passive *listener* receives a notification for every input received from the *Input* component on the *input* channel, and a notification for every output received by the *Output* component on the *output* channel.

## 6 Example systems

Configurations of the system can be specified in a separate file, whose name can then be indicated in the top level *config.py* under *system_description_file* key. All example systems' configuration files are organized under *system_configs* directory. To launch the system once all configurations have been completed, we simply need to run

```
python main.py
```

### 6.1 Echo system

Echo system is a simple system which copies the input to the output. Therefore, it has no intermediate data processing and no required training. However, for demonstration purpose, all modes are setup as examples for the system.

The following table describes the components involved in this simple system in **model execution mode**:

| **Component** | | **Description** | **Note** |
|---|---|---|---|
| Domain Knowledge | | EmptyDomainKnowledge | An empty domain |
| Input | | StdInputDevice | Takes inputs from stdin |
| Preprocessing | | VoidProcessing | Does nothing to the input |
| Model | Preprocessing | VoidProcessing | |
| | Postprocessing | VoidProcessing | |
| | Internal | EchoAgent | Pipe the input to the output |
| Postprocessing | | VoidProcessing | Does nothing to the output |
| Output | | FileOutputDevice | Output to a file |
| Listeners | Scoring | SampleScoring | Sends sample notifications |
| | Logging | LoggingListener | Log all pubsub notifications to file |

Table 1: Setup for the echo system in execution mode

The above components are specified in a simple manner in the main configuration object. Each component is identified by a key and specified within a dictionary including the name of the class, initialization parameters and runtime configurations if applicable.

1. Implementation class is specified using the key *class*. The class must be an appropriate python class for each component.

2. Configurable parameters in the object instantiation can be specified with *args* key in the specification dictionary. The Output class with *FileOutputDevice* is a good example for this.

3. Runtime configuration parameters for the object can be specified with *config* key in the specification dictionary. Since the **Echo system** is simple, we do not need any runtime configuration.

## 6.2 HRED model

HRED model is an adaptation of the following paper into the system: Building End-To-End Dialogue Systems Using Generative Hierachical Neural Network Models. The following table describes the components involved in this simple system in **model execution mode**:

The following table describes the components involved in this simple system in **model execution mode**:

| **Component** | | **Description** | **Note** |
|---|---|---|---|
| Domain Knowledge | | EmptyDomainKnowledge | An empty domain |
| Input | | StdInputDevice | Takes inputs from stdin |
| Preprocessing | | HredPreprocessing | Split input sentence into tokens |
| Model | Preprocessing | Model specific | Add model specific tokens to the utterance. |
| | Postprocessing | Model specific | Remove speaker tokens |
| | Internal | HredAgent | Internal implementation of HRED |
| Postprocessing | | VoidProcessing | Does nothing to the output |
| Output | | FileOutputDevice | Output to a file |
| Listeners | | LoggingListener | Log all pubsub notifications to file |

Table 2: Setup for HRED system in execution mode

The main configuration is very similar to that of Echo Agent. A few modules were replaced to have the new system in place.

## 6.3 POMDP model

We adapt the POMDP model implementation from this Github repository. There are only few changes in system configurations. The following table describes the components involved in this simple system in **model execution mode**:

| **Component** | | **Description** | **Note** |
|---|---|---|---|
| Domain Knowledge | | VoiceMailPomdpDomainKnowledge | Specifying states, observations and actions |
| Input | | StdInputDevice | Takes inputs from stdin |
| Preprocessing | | VoidPreprocessor | |
| Model | Preprocessing | VoidProcessing | |
| | Postprocessing | Model specific | Format action taken and expected rewards |
| | Internal | POMDPAgent | Internal implementation of POMDP |
| Postprocessing | | VoidProcessing | Does nothing to the output |
| Output | | FileOutputDevice | Output to a file |
| Listeners | | LoggingListener | Log all pubsub notifications to file |

Table 3: Setup for POMDP system in execution mode

The main configuration file only has few modifications compared to that of EchoAgent and HRED.

## 6.4 Goal oriented models

We implemented a simple rule-based to demonstrate the system with a goal oriented model. The example system contains a generic agent that first identifies the user intent (from asking for *address* and asking for *name*). The system then will query the domain knowledge for the slot information for the identified intent and execute the policy accordingly. The following table describes the components involved in this system in **model execution mode**:

| **Component** | | **Description** | **Note** |
|---|---|---|---|
| Domain Knowledge | | GoalOrientedDomainKnowledge | Specifying slots information for known domains |
| Input | | StdInputDevice | Takes inputs from stdin |
| Preprocessing | | VoidPreprocessor | |
| Model | Preprocessing | VoidProcessing | |
| | Postprocessing | VoidProcessing | |
| | Internal | PersonalInformationAskingModel | Intent disambiguation and execution policies |
| Postprocessing | | VoidProcessing | Does nothing to the output |
| Output | | FileOutputDevice | Output to a file |
| Listeners | | LoggingListener | Log all pubsub notifications to file |

Table 4: Setup for goal oriented system in execution mode

## 6.5 Mturk integration

### 6.5.1 Response collection

The following table describes our setup for mturk to collect response given a set of contexts:

| **Component** | | **Description** | **Note** |
|---|---|---|---|
| Domain Knowledge | | EmptyDomainKnowledge | An empty domain knowledge |
| Input | | ContextFetchingInputDevice | Fetches contexts from stdin |
| Preprocessing | | VoidPreprocessor | |
| Model | Preprocessing | VoidProcessing | |
| | Postprocessing | Model specific | Format action taken and expected rewards |
| | Internal | MturkCollectionAgent | Receive responses from the server and place them as output of the system |
| Postprocessing | | VoidProcessing | Does nothing to the output |
| Output | | FileOutputDevice | Output to a file |
| Listeners | DatabaseResponseListener | | Collect responses and write them to database |
| | LoggingListener | | Log all pubsub notifications to file |

Table 5: Setup for mturk response collection

### 6.5.2 Scoring

The following table describes our setup for mturk to score responses produced by the HRED model:

| **Component** | | **Description** | **Note** |
|---|---|---|---|
| Domain Knowledge | | EmptyDomainKnowledge | An empty domain knowledge |
| Input | | ContextReceivingInputDevice | Fetches contexts received from users |
| Preprocessing | | MturkScoringPreprocessingAdapter | An adapter that adds information to track scoring on top of a plain VoidProcessing unit |
| Model | Preprocessing | VoidProcessing | |
| | Postprocessing | VoidProcessing | |
| | Internal | MturkScoringPreprocessingAdapter | Wrap around any regular agent (HRED in this case) to track scoring information |
| Postprocessing | | MturkScoringPostProcessing | Another wrap around to maintain scoring information |
| Output | | NotifiedResponseOutputDevice | Remove added scoring information and output the regular output to file |
| Listeners | DatabaseScoringListener | | Collect contexts from users as well as scorings and write them to database |
| | LoggingListener | | Log all pubsub notifications to file |

Table 6: Setup for mturk response scoring