BABEŞ-BOLYAI UNIVERSITY

Faculty of Economics and Business Administration

Business Management and Distributed Computing

Dissertation

Graduate,

Vasile-Ionuț-Remus **IGA**

Supervisor,

Prof. Gheorghe Cosmin **SILAGHI**, PhD

2023

BABEŞ-BOLYAI UNIVERSITY

Faculty of Economics and Business Administration

Business Management and Distributed Computing

Dissertation

Ontology-driven dialogue simulator for generating task-oriented dialogue datasets

Graduate,

Vasile-Ionuț-Remus **IGA**

Supervisor,

Prof. Gheorghe Cosmin **SILAGHI**, PhD

2023

Abstract

Building task-oriented dialogue systems for solving particular tasks in different domains represents a challenging research objective, especially when there is a lack of specific datasets for training deep learning components. Our approach aims to bootstrap the development of a conversational agent by generating a dataset that contains an arbitrary number of dialogues for training any neural network of one's choice. We build a Machine-to-Machine (M2M) system, with three main components: a prompt generator, a user simulator, and a task-oriented dialogue system (TODS). With the help of semantic technologies, the domain-scope knowledge is mapped under an ontology, and the dialogue context is represented as a local knowledge graph, while pre-defined rules transform text templates into natural language responses. The final metrics obtained highlight the benefits of the aforementioned framework.

Contents

Abbreviations

DST,	dialogue state tracking
KG,	knowledge graph
M2M,	machine to machine
NLG,	natural language generation
NLU,	natural language understanding
POL,	dialogue policy
SOTA,	state-of-the-art
TOD,	task-oriented dialogue
TODS,	task-oriented dialogue system

List of tables and figures

1. Introduction

The ability to converse freely in natural language is one of the hallmarks of human intelligence and is likely a requirement for true artificial intelligence [1]. This ability is implemented by the domain’s scientific research in two ways: task-oriented dialogue systems and chatbots. A task-oriented dialog system aims to assist the user in completing certain tasks in a specific domain, such as restaurant booking, weather query, and flight booking, which makes it valuable for real-world business [2]. Chatbots are systems designed for extended conversations, set up to mimic the unstructured conversations or ‘chats’ characteristic of human-human interaction, mainly for entertainment, but also for practical purposes like making task-oriented agents more natural [3]. Therefore, such systems can benefit any organization that encounters repetitive tasks or ones that do not require human intervention.

From a text processing point of view, there are two main tasks to solve:

1. Understanding of the user's utterance
2. Generation of a system response.

[2] and [3] propose two modules for solving the first task, which are natural language understanding (NLU) and dialogue state tracking (DST), while for the second one, they suggest dialogue policy (POL) and natural language generation (NLG). Older solutions were rule-based, more exactly a combination of ‘if’ branches to fit as many user utterances as possible, but as the task becomes harder to solve, more rules are necessary, thus increasing the number of different user utterances to be matched. Modern solutions imply neural networks, which automatically extract the desired information from text. This leads to an important advantage, as rules are no more mandatory to fit the whole possibilities’ space, therefore increasing the capacity of gathering precious information from unstructured data. Similar systems can be found in digital assistants, such as Siri, Alexa, Google Now/Home, Cortana, etc. [3]. Although significant differences can be seen between the two approaches, they are compatible, which leads to a hybrid architecture capable of maximizing the benefits of both methods.

Building a conversational agent supposes either the connection of the four mentioned components or a single neuronal network capable of solving all tasks (also called the end-to-end model), this being the current state-of-the-art approach [2].

More recently, pre-trained neural network models have emerged, which ease the training of a customized model by enabling transfer learning, a concept of using the pre-acquired knowledge as a starting point for a specific model, which now only requires fine-tuning for the desired task. Such models are BERT [12], T5 [13], or GPT-3 [14].

Natural Language Understanding is an essential component that typically includes the intent classification and slot-filling tasks, aiming to form a semantic parse for user utterances. Intent classification focuses on predicting the intent of the query while slot-filling extracts semantic concepts [11]. An example would be 'Insert a project with code as 123 and name as Robot Assistant.', where the intent can be 'insert' and slots are key-value pairs, such as 'code' = '123' and 'name' = 'Robot Assistant'. All intents and slots are defined by a well-structured ontology.

Most SOTA models for NLU are fine-tuned on top of BERT [6], [11], [15], T5 [4], [7], and BART [5]. Classical approaches use neural networks of type Long Short-Term Memory (LSTM) or Convolutional Neural Network (CNN) [9], [10]. Datasets of reference are SNIPS [16], ATIS [17], BANKING77 [18], or RESTAURANTS8K [19], while the most used metrics are intent detection accuracy and F1 score for slot-filling [4], [5], [6], [11], [15].

Dialogue State Tracking provides the core of a TOD system [10], estimating the user's goal in each time step by taking the entire dialog context as input [2]. The state translates to a series of slots (from the whole dialogue, not just the last utterance, as opposed to NLU) and (not mandatory) the last system's action. Continuing the example from above, we pretend the system responds with 'There are more required parameters that you need to insert: class, status. There may be some optional ones too: manager.', and the user adds 'class is Python, status is done and manager is John.'. The state would translate to 'requireParams(code = 123, name = Robot Assistant, class = Python, status = done, manager = John)', where 'requireParams' asked the user to insert the other mandatory parameters and the information between the parenthesis holds all the mentioned slots throughout the dialogue. This output conditions the next system action, determined by the POL component.

More recent approaches transform the state into a local graph, where each user utterance goes through a model that generates a *program*, representing the system's response [8]. The last approach is of more interest to us, as we experienced interaction with knowledge graphs and can leverage our expertise.

SOTA models are built using T5 [4], BART [5], or BERT [6], while classical approaches make use of Jordan-Type Recurrent Neural Network (RNN) and CNN feature extractor [10]. Important datasets used for training are MultiWOZ [20], CamRest676 [10], Frames [21], and TaskMaster [22]. The most frequently used metrics are Joint Goal Accuracy (JGA), precision, recall, or F1 score [4], [5], [6], and [10].

Dialogue Policy decides what the system should do next based on the dialogue state, but it can also integrate database or intent and slot detection search results. Having the above example in mind, the action generated by POL might be ‘confirmParams’, which asks for user confirmation of the correctness of all parameters inserted until that point of discussion. Together with other internal variables, the NLG component will translate the system action into natural language.

BERT [6], T5 [4], [7], or BART [5] are the foundation of SOTA models for POL tasks, while other methods use Reinforcement Learning, planning, or a combination of LSTM with attention mechanisms [2], [3], [9], [10]. Well-known datasets for this task include Schema Guided [23], MSR-E2E [24], Frames [21], or TaskMaster [22], using inform rate, success rate, F1 score, accuracy, or recall for measuring the model’s performance [4], [6], [7], [9], [10].

Natural Language Generation returns the natural language response form of the system’s action. It is often modeled in two stages, content planning (what to say), and sentence realization (how to say it) [3]. Planning can be done by POL, leaving sentence realization on NLG. To increase the number of possible responses, one might use the delexicalization technique, which involves substituting the slot values with the names of the slots. Next, a relexicalization process introduces the slot values back to phrases. This ensures that the model does not learn a phrase with a specific value but with a possible slot. The last piece of the ongoing example includes a response for ‘confirmParams’, under the form of ‘I need you to confirm the values for the following slots: <slots>’, where <*slots*> will be substituted with the real values.

Pre-trained models are the choice for SOTA models, including T5 [4], BART [5], BERT [6], and GPTs [7]. Classical procedures leverage attention mechanisms, LSTMs, or encoder-decoder architectures [3], [10]. MultiWOZ [20], Schema Guided [23], MSR-E2E [24], Frames [21], or TaskMaster [22] are frequently used for training, while the performance is computed using BLUE score, exact match rate, or k-to-100 accuracy [4], [5], [6], [10].

Regardless of the architectural choice, one needs specific data to fine-tune a model for solving tasks from a certain domain. A model is only aware of the concepts learned during training, therefore the quality of the used datasets massively influences the capacities of a model. In the TOD systems domain, such sets of data have to be gathered from use cases met in real-life scenarios, which are not usually available online. Therefore, one needs to build its dataset, a task that requires plenty of resources, such as staff, money, time, domain expertise, etc. Three main strategies can be employed for building a custom dataset [20]:

- **Machine-to-Machine**, where two simulators generate annotated dialogues, using built-in rules; has the advantage of controlling and matching a large number of possible cases, without the need for human intervention, with low resources. The disadvantage is the possible lack of naturalness in the generated dialogues.
- **Human-to-Machine** is an intermediary solution, that requires an already functional dialogue system. It labels the conversation between users and the machine but is limited by the understanding capacity of the initial system, introducing bias in the collected data. Such an approach is suitable for already heavily tested and expert-verified systems.
- **Human-to-Human**, arguably the best solution, implies the dialogue between two humans, which makes use of an interface to automatically label the chat between themselves. Although it ensures the most natural conversations, it can generate costs of different types, such as: developing a platform to be used by workers, training for using the system, payments for their work, etc. In many cases, one might not have the necessary resources to develop such systems, and many find themselves blocked at this stage.

Until recently, knowledge graphs were a technology that was not so popular among the developers of conversational agents. With their ability to represent information using concepts and relationships, KGs can enhance the study of natural language. Moreover, they help analyze text by providing background knowledge in a humanized way, enriching it with the facts extracted from natural language utterances.

To train a neural network competent enough to solve a specific task, datasets covering many of the possible real-life scenarios are needed. H2H systems can be expensive, while H2M requires an already existing, well-functioning conversational agent to generate quality datasets. Therefore, in our thesis we construct a M2M system, to bootstrap the creation of a conversational agent by generating a dataset that contains an arbitrary number of dialogues for training any neural network of one's choice. It has three main components: a prompt generator, a user simulator, and a task-oriented dialogue system. We integrate semantic technologies, by mapping the TODS's knowledge under the form of an ontology, while the dialogue context is represented by a local knowledge graph. Pre-defined rules transform text templates into natural language responses.

The prompt generator is the catalyst of the entire system. It generates a prompt that is considered the overall structure of a dialogue scenario. The prompt is fed into the user simulator which will try to solve each mentioned task.

The user simulator mimics the behavior of a user in specific situations by mapping the intent and system action into natural language utterances. It has a rule-based design, merged with probabilities of different user intents.

For the TOD system, we combine the modular (or pipeline) approach of building a task-oriented dialogue system with semantic technologies, to increase the understanding of natural language. Also, we use the rule-based method with templates for providing user support with the implemented procedures for knowledge management. The system can understand provided intents and has support for Create-Retrieve-Update-Delete (CRUD) procedures on a given knowledge base. It requires an ontology, which holds the information about an environment in the form of concepts and relationships, considered as the general knowledge of the system. An important aspect of our research is the integration of multiple threads of discussion within the same dialogue. Figure 1 shows a summarized version of the system's architecture.

The thesis is structured as follows: Chapter 1 presents related work which influenced our scientific research, Chapter 2 describes the architecture of the M2M system, Chapter 3 highlights and discusses the obtained results, and Chapter 4 concludes the dissertation. The supporting materials are to be found within the Appendices.

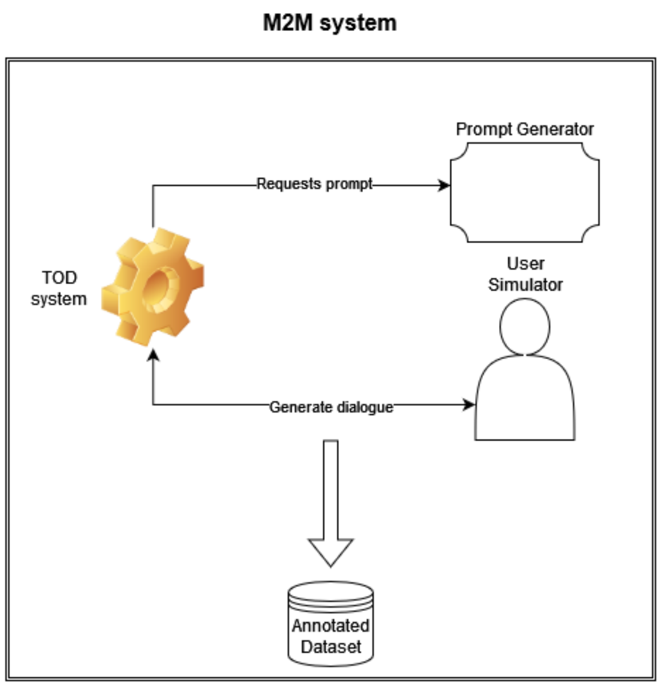


Figure 1. Summarized diagram of the M2M system architecture.

2. Related Work

In the domain of conversational agents, dialogues are defined as a series of turns, where each turn is composed of a user utterance and a system response, under textual form. An optional layer of speech recognition can be added, to detect audio dialogues. We intend to make conversations look as natural as possible, thus we permit each conversation to have multiple discussion threads, on topics defined by the provided ontology. The general objective of a dialogue system is to generate a response for each user entry. Its specific objective is to help users in accomplishing some requested procedures. To generate a personalized dataset for training different neural models, a dialogue simulator can be used.

[25] uses an M2M system to generate dialogue data for a domain described by a relational database. They design an agenda-based user simulator together with a domain-agnostic rule-based dialogue manager that interacts with each other to generate the desired dataset. An extra step is taken, as they fed the dataset into an extended version of RASA[1], which trains all the core components of a conversational agent. [26] propose an M2M framework to rapidly bootstrap the generation of end-to-end models, by combining automation and crowdsourcing. In the first phase, they use a user simulator and domain-agnostic system to generate dialogues, using scenarios and outlines, ideas that inspired our approach.

[8] are the first ones to use semantic technologies for state tracking in task-oriented dialogue systems. It demonstrates not only that knowledge graphs can replace classic state tracking approaches in dialogue systems, but prove that the system gains benefits from integrating them. Having this in mind, we build our research around their work, by using KGs for state tracking. Extending their findings, the dialogue simulator builds its knowledge around a given ontology (which is represented as a KG). Similar to us, [26] present a chatbot for fashion brands, based on the semantic web. The ontology-driven chatbot model can answer questions from users interested in the latest information about fashion brands in Pakistan. They use some technologies like ours, for example, Resource Description Framework (RDF) for writing the ontology, or SPARQL for querying the knowledge graphs.

[1] https://rasa.com/

[27] and [28] leverage the use of KGs as the system's knowledge but they take a step forward and integrate the obtained knowledge into Deep Learning models capable of generating user responses. [29] present a somehow similar idea of building a local conversational graph, which is later integrated with a BERT model that learns to answer using the elements of the KG, in an end-to-end fashion. A similar approach is given by [30], who creates a testbed to check the strong and weak points of end-to-end dialog systems in goal-oriented applications. They use a simulator for generating specific datasets, which is designed to manipulate both natural language and knowledge graphs. The user and bot utterances are created using natural language patterns which are integrated with KG entities to form different phrases, similar to how we generate responses.

3. Methodology

The section describes each element of the M2M system, regarding their structure and goals. The system has three components: the prompt generator, the user simulator, and the TOD system.

Python 3.10 is used to write the underlying code, the ontology is described in RDF, which is the same format for the local knowledge graph, with the help of Turtle notation. The local KG that is maintained during each conversation and the global KG are managed with the RDFLib[2] library. The global KG is persisted in an external file at the end of each running experience. The queries regarding the local and the global KGs are written in SPARQL. The sample pools for parameters' values randomization are provided in text files. The generated dataset is exported in a JSON file, while some statistics about the simulation are kept in text documents.

3.1 Prompt Generator

A dialogue between persons usually follows a path described by one's intentions and others' responses. It is no different when creating machine-to-machine simulators, as a starting scenario must be designed, to bootstrap the conversation. The prompt generator is integrated into the TODS, which calls for a prompt to be produced for each scenario.

[2] https://rdflib.readthedocs.io

In our case, the starting point is represented by a simple, yet compact structure. Each dialogue scenario takes the form of a standard template: *say hello. do procedure. say goodbye*. Breaking down its structure, there are two possible types of intents: simple, which does not require any parameter (preceded by *say*), or complex, which may have parameters (preceded by *do*). The whole system supports 12 possible user intents: hello, goodbye, thank, insert, select, delete, update, agree, disagree, remove, cancel, and switchEntity. Out of all, only six can be used for building scenarios, the others are generated throughout the dialogue between the user simulator and TODS. *Hello* and *Goodbye* are considered simple intents and are always present in a scenario. *Insert, Update, Delete,* and *Select* are the complex intents that may appear in a scenario for a limited amount of times.

Update, Delete, and Select can appear one or two times in a scenario, as they are procedures that do not create other discussion sub-threads. The interval was set to prevent too long conversations while holding a minimum acceptable number of turns per dialogue (the worst-case scenario has four turns).

Insert can be generated only once in a starting scenario, but it is a procedure that may spawn sub-threads when the discussed instance needs to be connected to other instances that do not exist yet (therefore, a sub-thread is started for inserting that instance) or at a random point of conversation to assure diversity of possible dialogue paths.

Each scenario can have only one of the four procedures. All procedures have a predefined probability of being included in the prompt, as follows:

- 40% for Insert
- 30% for Select
- 20% for Update
- 10% for Delete.

Insert has the highest probability, because of one main reason: as the starting knowledge graph is almost blank (having only eight instances), it needs to populate it, so the other procedures have data to interact with. Therefore, it has a greater percentage than the others.

Select is the most common procedure a user might do; thus, it is the second in terms of probabilities.

Update and Delete were differentiated by the fact that the latter erases information from the KG, therefore other procedures might not interact with any instances, which is rarely a real-world scenario. Having this in mind, Delete has a lower probability than Update, which is also the lowest among all.

Each procedure has a lot of possible ways of being formulated in natural language, but few main cases can be established (each prompt is a shorter version of its natural language counterpart, thus may not respect semantic/syntactic rules).

Insert has one main approach, which is *do insert a entity_type [with parameter value parameter value etc.]*. It focuses on inserting an instance of a specific type, which may or may not include parameters from the start. Then, the user simulator and the TOD system will develop a conversation and grow different possible paths, including adding/removing parameters, spawning secondary threads, canceling ongoing procedures, and more.

Update, Delete and Select follow three principal schemes:

- do procedure a entity [with parameter value parameter value etc.] [where parameter value parameter value etc. (only for update)]
- do procedure all entities [with parameter value parameter value etc.] [where parameter value parameter value etc. (only for update)]
- do procedure ID [with parameter value parameter value etc. (only for update)].

The first case focuses on a situation where the user wants to interact with a single instance of a specific type but does not know for sure which one. The user can opt for adding filters for different parameters. The second case works similarly, but it affects all instances of a type. The last case involves interaction with an instance mentioned by ID. To simulate different scenarios, each case can generate wrong entity types/IDs with a probability of 10%. This leads to misunderstandings between the user and TODS, which can happen in real-life situations. Each procedure has its probability for a case to happen.

Select has 50%, 40%, and 10% for the three cases, with a 50-50% probability of having/not having parameters in the first two scenarios. Update is a special procedure, in the sense that it may have filter parameters (where clause) or the new values to be updated (with clause). It has 80%, 10%, and 10% for the three cases, with a 90-10% probability of having/not having parameters (with clause) in the first two scenarios. The where clause has a 50-50% probability of being included. Delete has 88%, 2%, and 10% for the three cases, with a 90-10% probability of having/not having parameters in the first two scenarios; the second case has a very low probability, to protect the knowledge graph from being completely deleted (a very rare real-life situation).

An example is shown in the figure below, where a prompt is built with the standard template. In the middle, the generated procedure is an Insert, from which the dialogue is going to evolve in ways defined by the probabilities from the user simulator.

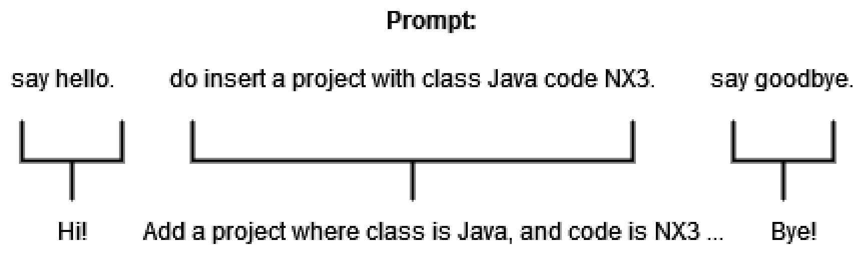

Prompt:
say hello.
do insert a project with class Java code NX3.
say goodbye.
Hi!
Add a project where class is Java, and code is NX3 ...
Bye!

Figure 2. An example of a prompt created by the Prompt Generator.

3.2 User Simulator

Another crucial component of an M2M system is the user simulator, which mimics the behavior (to an extent) of a real person. A provided list of intents and ontology represents its knowledge and the probabilistic paths are its behaviour. In our case, the user simulator is an integrated component of the TODS, which calls for an utterance at each step, until the dialogue finishes.

The simulator starts by breaking down a prompt into singular tasks. It uses built-in rules to separate each task from one another, as presented in the pseudocode:

```
START
  Separate each task by ‘.’ (dot);
  FOR each task
     Initiate a local dictionary;
     The intent is always the second word, therefore get the second word from the task
     as ‘intent’;
     IF the first word is ‘say’
                   Go to the next task;
     ELSE IF the first word is ‘do’
          IF the second word is in [‘a’, ‘an’, ‘all’]
                   Get the third word as ‘entity’;
          ELSE IF not on the list
                         Get the second word as ‘instance’;
                END_IF
     END_IF
```

```
        IF the length of the phrase is greater than 4 and the fifth word is ‘with’
                Get the list of parameters;
            END_IF
        END_FOR
END
```

For update, there is a supplementary condition for getting the filter parameters. More exactly, if the word ‘where’ is found, everything that follows falls into the filter category. All the parameters after ‘with’ are the new values to put in place of the existing ones.

Each task represents a user intent, from the list provided at initialization time. When the user simulator is first called, it loads the first task and always moves to the next one (as the first one is Hello intent). Then, it activates the complex intent and settles down for it until the TOD system prompts that it can move forward to the next one. This is done by matching different TODS actions for each complex intent. As all prompts finish with a Goodbye intent, the user simulator stops loading any other after reaching it.

There are four complex intents, the procedures described above. Each has different system actions to react to, with specific probabilities. Regardless of the ongoing procedure, the TODS may generate the default action, to which the simulator reacts by moving to the next task.

Select has four possible actions from the TODS: *showSelect*, *wrongSelect*, *wrongFormatSelect*, and *removeParams*.

showSelect is the action that prompts to the user the list of instances that map the desired filters, or the action which signals the success of the query. In response to it, four possibilities exist:

- 42% chance to move to the next task, as the current one is successful
- 32% chance to add supplementary parameter(s) to the query
- 18% to remove parameter(s) from the query
- 8% chance to cancel the current query and move to the next task.

wrongSelect and wrongFormatSelect are returned when the query does not find any instances, considering different circumstances, to which the simulator reacts by moving to the next task.

removeParams is the TODS’s confirmation of successfully removing the parameter(s). The simulator will then ask for continuing the procedure.

Delete has five possible actions from the TOD system: *showDelete, wrongDelete, confirmDelete, dependencyDelete,* and *removeParams.*

showDelete is the action that prompts the user to the list of instances that map the included filters and asks for confirmation of the instance(s) to be deleted. In response to it, five possibilities exist, each with a probability of:

- 49% to delete an instance:
    - 80% to choose one from the list
    - 20% to choose one outside the list.
- 24% to delete all the instances from the list
- 12% to add supplementary parameter(s) to the query
- 10% to remove parameter(s) from the query
- 5% to cancel the current query and move to the next task.

The simulator reacts by moving to the next task for wrongDelete (the query does not return any instance, considering different circumstances), dependencyDelete (other instances are linked to the instance to be deleted), and confirmDelete (the query was successful).

removeParams is the TODS's confirmation of successfully removing the parameter(s). The simulator will then ask for continuing the procedure.

Update has nine possible actions from the TOD system: *dependencyUpdate, wrongLiteralDataFormat, showUpdate, chooseEntity, chooseReject, confirmUpdate, wrongUpdate, preExistingEntityCancel,* and *removeParams.*

dependencyUpdate (the instance is to be updated with values that refer to non-existing instances in the KG, where pre-existing ones are mandatory) and wrongLiteralDataFormat (some values are in the wrong format for the literal parameters) have six possibilities of response, each with a probability of:

- 80% to insert a new value for one or more highlighted parameters
- 8.8% to cancel the procedure
- 4.6% to add new value(s) for any parameter(s)
- 2.2% to add new value(s) for any filter parameter(s)
- 2.2% to remove parameter(s) from the query
- 2.2% to remove filter parameter(s) from the query.

showUpdate is the action taken by the TOD system when it asks for confirmation of which instance to update, to be chosen from the provided list. It has seven possible paths, with a chance of:

- 45% to update an instance, with:
    - 80% to choose one from the list
    - 20% to choose one outside the list.
- 35% to update all the instances from the provided list
- 8.8% to cancel the procedure
- 4.6% to add new value(s) for any parameter(s)
- 2.2% to add new value(s) for any filter parameter(s)
- 2.2% to remove parameter(s) from the query
- 2.2% to remove filter parameter(s) from the query.

chooseEntity (the user is required to choose an instance from the lists provided for specific parameters) and chooseReject (the user did not choose from the list that has been given for each parameter) have six possibilities of response, each with a probability of:

- 80% to choose an instance for one or more highlighted parameters, with:
    - 80% to choose one from the list
    - 20% to choose one outside the list.
- 8.8% to cancel the procedure
- 4.6% to add new value(s) for any parameter(s)
- 2.2% to add new value(s) for any filter parameter(s)
- 2.2% to remove parameter(s) from the query
- 2.2% to remove filter parameter(s) from the query.

The simulator reacts by moving to the next task for wrongUpdate (the query does not return any instance, considering different circumstances), preExistingEntityCancel (the instance to be updated coincides with an existing one in the knowledge graph), and confirmUpdate (the query was successful).

removeParams is the TOD system's confirmation of successfully removing the parameter(s). The simulator will then ask for continuing the procedure.

Insert is the most elaborated intent, as it has 13 possible TODS actions, including *requireParams, switchEntity, confirmParams, preExistingEntityCancel, cancelProcedure,* *wrongLiteralDataFormat*, *chooseEntity*, *chooseReject*, *removeParams*, *askStep*, *confirm*, *wrongEntity* and *reject*.

requireParams (user needs to provide at least the mandatory parameters for the currently discussed instance) and switchEntity (user switched to another active thread of discussion) have three possible responses, including:

- 93% probability to add new parameter(s)
- 5% probability to remove parameter(s)
- 2% probability to cancel the on-going procedure.

confirmParams signals that all the mandatory parameters were given by the user for the discussed instance, having one of the five responses, with a probability of:

- 90% agree with the insertion
- 5% disagree with the insertion
- 4.25% add new parameter(s)
- 0.5% remove parameter(s)
- 0.25% cancel the ongoing procedure.

preExistingEntityCancel (TOD system cancels the insertion of the instance because it already exists in the KG) and cancelProcedure (TODS confirms the user's intention of canceling the current process) both have reactions conditioned by the existence of other already started threads of discussion. More exactly:

```
START
    IF there is at least one existing thread
        IF there are more threads of discussion, switch to
            80% an already existing thread;
            20% a non-existing thread;
        ELSE IF there is only one active thread
            25% probability to add new parameter(s);
            25% probability to remove parameter(s);
            50% probability to cancel the ongoing procedure;
        END_IF
    ELSE IF there is no active thread
        50% spawn a random insert process;
        50% move to the next task;
    END_IF
END
```

wrongLiteralDataFormat (some values are in the wrong format for the literal parameters) triggers four cases:

- 90% chance to add new values for one or more highlighted parameters
- 2.5% probability to add new parameter(s)
- 2.5% probability to remove parameter(s)
- 5% probability to cancel the on-going procedure.

chooseEntity and chooseReject have four possibilities of response, each with a probability of:

- 90% to choose an instance for one or more highlighted parameters, with:
    - 80% to choose one from the list
    - 20% to choose one outside the list.
- 2.5% probability to add new parameter(s)
- 2.5% probability to remove parameter(s)
- 5% probability to cancel the on-going procedure.

removeParams only requires confirmation of continuing the ongoing procedure.

askStep (the response of the TODS when the user disagrees with the insertion of an instance) provides three ways of reaction, including:

- 45% probability of spawning a new random process
- 30% probability of canceling the ongoing one
- 25% probability to keep the execution of the current one.

wrongEntity (triggered by the switch to a wrong/non-existing thread of discussion) is solved using two cases with chances of:

- 80% to switch to another thread, with:
    - 90% probability of switching to an existing thread
    - 10% probability of switching to a non-existing thread.
- 20% to move to the next task.

reject (the instance is to be inserted with values that refer to non-existing instances in the KG, for parameters that require pre-existing ones) matches five routes, each with different odds, including:

- 90% of times starts new thread(s) of discussion that focus on inserting the non-existing values
- 8% of the time adds another value for one or more highlighted parameters
- 0.5% probability to add new parameter(s)
- 0.5% probability to remove parameter(s)
- 1% probability to cancel the on-going procedure.

confirm (TODS confirms the insertion of the instance and might switch to another thread if exists) has possible solutions conditioned by the existence of other already started threads of discussion. More exactly:

```
START
    IF there is at least one existing thread
        IF the thread was spawned by another one
            IF there are no other sub-processes to be spawned
                Confirm the parameters for the instance in the main thread;
            ELSE IF there are other sub-processes to be spawned
                Start a new sub-process;
            END_IF
        ELSE IF the thread was not spawned by another one
            50% confirm the insertion of the current instance;
            20% switch to
                80% an already existing thread;
                20% a non-existing thread;
            7.5% probability to add new parameter(s);
            7.5% probability to remove parameter(s);
            15% probability to cancel the ongoing procedure;
        END_IF
    ELSE IF there is no existing thread
        move to the next task;
    END_IF
END
```

In the description of Insert, it is mentioned that the system spawns sub-processes (or sub-threads) of discussion. This refers to the ability of the user simulator to adapt to the conversation, by starting other threads to complete the main one. There are two types of spawning: random and dependent. Random spawning is used to simulate a least expected behavior from the user, by arbitrarily initiating sub-threads when askStep, preExistingEntityCancel, or cancelProcedure is generated by the TOD system. Dependent spawning occurs when the TODS produces either reject or confirm action, as the user simulator takes the initiative of starting sub-processes to insert the non-existing instances introduced for parameters that require pre-existing ones in the KG.

Random and dependent spawning work in a similar fashion: both insert in the tasks list a new task immediately after the currently discussed one. Dependent spawning has to do extra work, by inserting into the current task a new key (called 'to_spawn'), that holds the parameters requiring instances in the KG. In this way, the simulator can check whether a task spawned and makes sure to re-execute it after the completion of all sub-threads.

All probabilities are designed to respect two principles:

1. Follow the normal expected behavior as much as possible
2. Leave room for unexpected turns in the dialogue.

Each probability will be adjusted after real-life deployment of the dialogue system which will show how humans follow the path of dialogue.

3.3 TOD System[3]

The TODS is the main component that holds together the entire dialogue simulator. Its role is to simulate the behavior of a system that helps a human user to solve specific tasks. In our case, it enables four procedures, known under the CRUD acronym (Create-Retrieve-Update-Delete).

Its structure respects the classical modular approach of building a task-oriented dialogue system but integrates semantic technologies, represented by knowledge graphs. We believe these graphs can increase the understanding of natural language, as they internally map knowledge in the shape of concepts and relationships, closer to how humans think. Furthermore, they facilitate features such as reasoning which deduce new facts from existing

[3] The TOD system is extensively described in V.I. Iga, G.C. Silaghi, Ontology-based dialogue system for domain-specific knowledge acquisition, accepted in the 31st International Conference on Information Systems Development, Lisbon, Portugal, 30 Aug – 1 Sept 2023

ones, based on logical rules. The TOD system utilizes two KGs, one storing its knowledge (the ontology) together with all the correct instances built after communicating with a user, and a local dialogue graph, which maps the conversation between the system and the user, to track the context under a semantic form. All intermediary data is kept in the local graph until it is valid to be inserted in the general one. After each discussion, the local KG resets.

We use the rule-based approach with templates for providing user support with the implemented procedures for knowledge management. Figure 3 shows the four components and the main tasks done within each other.

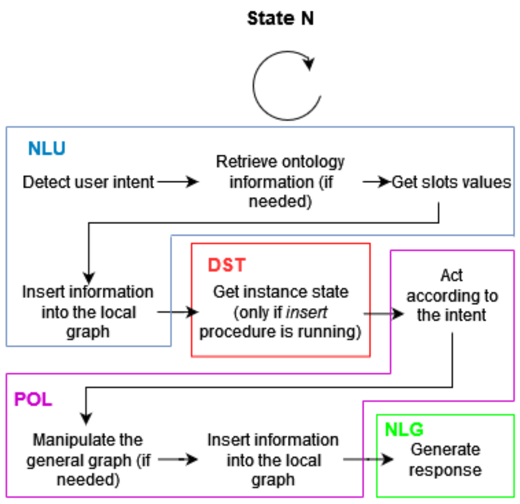

State N
NLU
Detect user intent
Retrieve ontology information (if needed)
Get slots values
Insert information into the local graph
DST
Get instance state (only if *insert* procedure is running)
Act according to the intent
POL
Manipulate the general graph (if needed)
Insert information into the local graph
NLG
Generate response

Figure 3. The pipeline architecture of the system.

The user starts a conversation with the system. Each conversation is divided into several turns, each consisting of a pair of user utterance and a system response. Based on the general KB, the system directs the conversation toward achieving one of the procedures enumerated above. The conversation ends either successfully or by any interruption. During the conversation, each validated procedure is sent from the local graph to the general KB. Figure 4 depicts a user-system conversation.

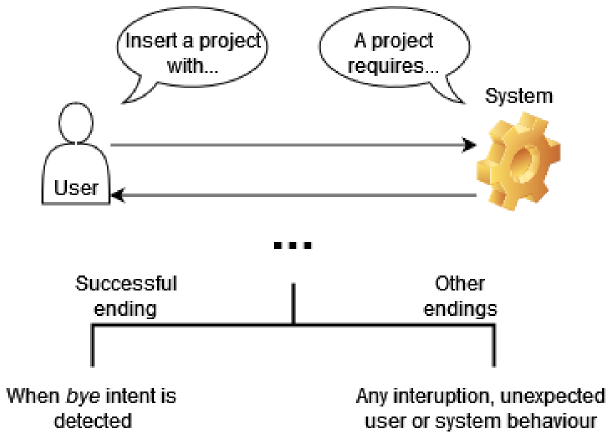


Figure 4. The conversation flow.

1.1.1 Ontology

The knowledge of the TODS is given by an ontology that contains concepts and relationships, specific to one or more domains. Each concept is defined by either complex properties that are other concepts, or by simple properties which cannot be extended anymore. Relationships are usually interpreted as parameters of concepts and may be mandatory or not. Its compulsoriness is given by the *owl:minQualifiedCardinality* property that defines the minimum number of concepts a relationship should be linked to. The mandatory relationships must be collected before an instance is considered well-defined.

Our system benefits from a compact ontology, consisting of three concepts (Status, Project, and Employee) and five relationships (hasStatus, hasCode, hasName, hasClass, hasManager). The main concept that links together all the other ones is Project, with five relationships, two of them being complex (hasStatus and hasManager). The last one is optional, therefore the connection between projects and employees may be omitted. Simple properties define their range as being a literal value, for example, string, integer, date, etc. The value does not need to refer to any existing instance in the KG and cannot be used as a starting concept. Figure 5 depicts the domain ontology used by the M2M system.

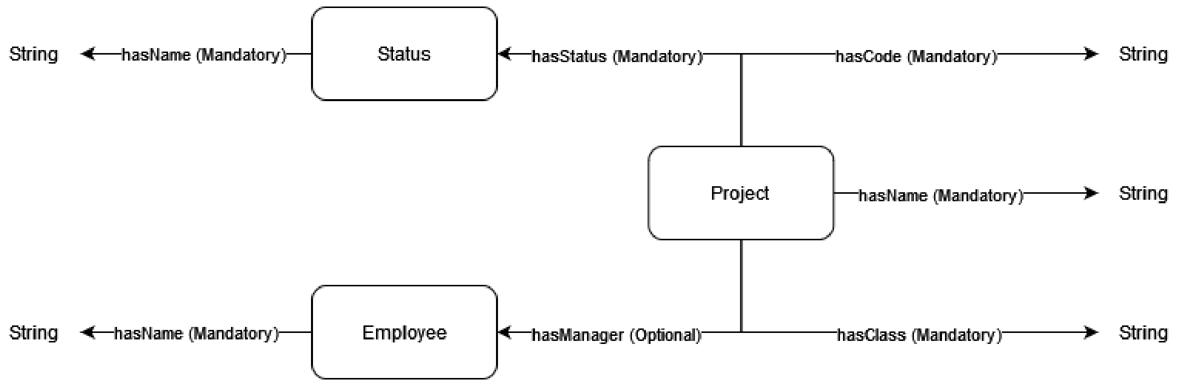


Figure 5. Domain ontology example.

1.1.2 Natural Language Understanding Module

NLU is the first module that processes the input from the user, solving tasks associated with intent detection, slot filling, and local graph management.

The module recognizes 12 intents*: hello, goodbye, thank, insert, select, delete, update, agree, disagree, remove, cancel, switchEntity,* and *default*, using keyword detection. Hello, Goodbye, and Thank are basic intents that enable the natural flow of conversation. Select, Delete, Update, and Insert are specific to the four implemented procedures. Agree and Disagree either confirm or postpones the ongoing Insert procedure, while Cancel stops it and switchEntity changes the active thread of conversation. Remove takes out parameters from the currently discussed instance. Default is the chosen intent when none of the others apply.

For slot filling, it first retrieves information regarding the related parameters of the current discussed instance by querying the ontology, followed by keyword detection of them in the user utterance. Both intent detection and slot filling are realized with the help of functions from the Utils class. Its final task is to update the local graph with the acquired information.

The information inserted in the graph is stored under the form of *User :detected_intent [anonymous node]*, where the *[]* node inserts the current turn together with all its slots and their specific values. At the same time, it instantiates a separate node (called the procedure or instance node) with specific relationships, which holds together the latest version of the discussed procedure or instance. It has a unique ID, assuring that the system is up to date with the conversation and can easily access information when needed.

1.1.3 Dialogue State Tracking Module

Traditionally, this module holds together the context of the discussion, with the most important aspects of it being stored. This approach has a noticeable disadvantage: managing many threads of discussion in the same dialogue becomes tedious work.

In our case, the context is stored by the local graph, while DST manages only the state of the instance being discussed. It does that by making use of one of the functions from the Query class, specialized for retrieving the state of one desired instance. It is locally saved in a dictionary by the TODS, for letting other components manipulate such information.

1.1.4 Dialogue Policy Module

This module can be considered the TODS's brain, as it reacts to the user intent. It manipulates both graphs if needed, either by retrieving, updating, deleting, or inserting new information. The policy module validates data before affecting the general KG and inserts important information into the local one regarding the discussion, under the form of *:System :action [anonymous node]*. In the *[]* node, the current turn is inserted, together with all collected information about the parameters. It also manipulates the procedure or instance nodes to update them to the latest state.

System actions could be one of the following: *hello, goodbye, welcome, wrongFormatSelect, wrongSelect, showSelect, confirmDelete, wrongDelete, dependencyDelete, confirmUpdate, wrongUpdate, dependencyUpdate, showUpdate, confirm,* *wrongLiteralDataFormat, cancelProcedure, askStep, reject, chooseReject, removeParams, requireParams, confirmParams, preExistingEntityCancel, chooseEntity, wrongEntity, switchEntity,* and *default.* They were designed to properly respond to all 12 possible user-detected intents.

1.1.5 Natural Language Generation Module

NLG module transforms the selected action into natural language. It has pre-built templates for all the above-mentioned actions. In each template, the system can dynamically insert generated information by replacing placeholders.

There are two types of actions:

- generics; simple actions that do not require placeholders (for example, Hello, Goodbye, or Welcome)
- dynamics; actions that manipulate or interact with the general and local KGs and may store information in local variables. Therefore, when generating natural language responses, each piece of information is processed and inserted in the text template by removing the placeholders.

An example of a template response is the next phrase: 'The <entity> will be inserted with the following params: <params>. Is it correct?'. <entity> will be replaced with the actual type of the currently discussed instance, while <params> gives its place to a list that contains the state of the instance.

1.1.6 Dealing with Multiple Discussion Threads

The ontology may contain concepts that are related to one another, thus the TODS needs to enable the insertion of the independent one, before the dependent instance.

To facilitate it, we generate different discussion threads for each instance to be inserted. Each time a thread is spawned, the ID of the former active instance is inserted in a LIFO-style list and a new instance with its specific ID becomes active. It is a recursive structure that may repeat until the user wants to. When a thread ends, the TOD system automatically switches to the latest in the list (thus the LIFO-style) to map the connection between the two instances. It also gives the user the chance to switch to another thread, if they want to. In this manner, we can manage multiple threads of discussion in the same dialogue, allowing the user to solve many tasks in the same round of chat.

Annex 1 details an example of a discussion with three threads, for a better understanding of this feature.

1.4 M2M System and Dataset Annotation

The main component of the system is the TODS, which integrates the other two in its structure. This architecture enables easy sharing of information between the three modules, without the need for an additional one to facilitate their communication. At the same time, the dataset annotation is done by the TODS.

The pseudocode and Figure 6 below describe the flow of the system when generating an arbitrary amount of scenarios. One needs to provide the supported intents, which is a dictionary with intents as keys and lists of keywords to be detected as values, the NLG templates, which is a dictionary that has TODS actions as keys and lists of possible natural language templates as values, the ontology and its namespace and the user simulator templates, used to generate user utterances. Next, it initializes the prompt generator and user simulator, and the M2M system is ready to be used. Then, the number of desired scenarios is inserted and for each:

- The TODS asks the prompt generator to provide a prompt
- It passes the prompt to the user simulator
- It requests a user utterance from the user simulator
- It passes the utterance through all of its modules and generates a response.

At each user utterance request, the TOD system passes key arguments to the user simulator, including its action, the procedures states, the ID and type of the discussed instance (if any), and the LIFO-style list (that holds the active threads). Each argument gives key insights into the state of the conversation. The user simulator does not need to receive the natural language response from the TODS, as it only needs its action to generate a user utterance.

The general flow of execution is described as pseudocode:

```
START
    Initialize TODS components;
        Get the supported intents, NLG templates, the ontology, and its namespace,
        and the user simulator templates;
        Initialize the prompt generator and the user simulator;
   Get the number of scenarios to be generated;
```

```
    FOR each scenario
        Create a prompt using the prompt generator and give it to the user simulator;
        WHILE the detected intent is different from goodbye
            Request the user utterance from the user simulator;
                Pass it the TODS action, procedures' states, the ID and type of
                the discussed instance (if any), and the LIFO-style list (that
                holds the active threads);
            Pass the utterance to the NLU module;
                Annotate the utterance with the discovered intent and slots;
            Call the DST module;
            Generate an action using the POL module;
            Map the action into natural language using the NLG module;
        END_WHILE
    END_FOR
END
```

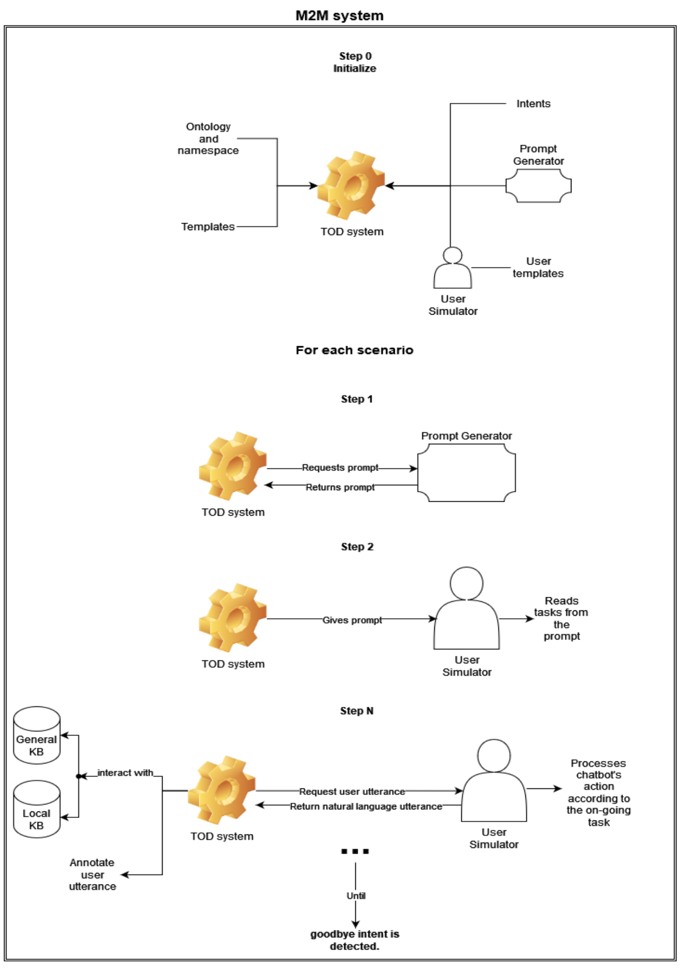


Figure 6. The overall architecture of the M2M system.

In the current version of the system, we annotate the dataset for the intent detection and slot-filling task, a key function of the NLU module. As we control the whole generation of the dialogues, the annotation can be done for any task we desire, but now the focus is on the NLU module, as we plan to integrate Deep Learning into our TODS. Only the user utterance is annotated, as the NLU module has to understand the user behavior and not the system's actions. Therefore, data is annotated in JSON format, as follows:

```
{
  'utterance ID': {
                    'text': 'user utterance',
                    'slots': {'slot name': 'slot value' etc.}
                    'positions': {'slot name': [start index, end index], etc.}
                    'intent': 'detected intent'
  }...
}
```

Each utterance has a unique ID, generated by the TOD system. Key elements are stored, such as:

- User utterance, under the 'text' key
- A dictionary of all slots detected by the NLU module
- The position of each slot in the text
- The intent of the user.

Annex 2 shows a part of a resulting dataset, for a better understanding of the structure.

1.5 Additional Important System Features

Besides the three main components, the M2M system makes use of additional important features. Each one focuses on a specific task, helping the overall system to accomplish its goals.

1.1.1 Utils Class

Each component of the system may execute tasks similar to one another. To increase the quality of the overall code, this class holds together functions that are not necessarily specific to dialogue processing, but smaller activities, such as: escaping special characters, generating an ID for an instance or procedure, checking the validity of an ID, verifying the compulsoriness of parameters, tokenizing the user utterance, replacing the placeholders from a response template with concrete values from the KG, getting the ID/type of the currently discussed instance/entity, retrieving the parameter values from the user utterance, helping the annotation of a user utterance for a specific task, reading a random line from a file or formatting values before system response generation.

Some functions perform more important tasks than others, therefore a description of those is included.

getID(self) function generates a unique ID based on the current date and time. It takes the year, month, day, hour, minutes, seconds, and milliseconds values (resulting in 20 digits) and concatenates them with the type of the discussed instance/ongoing procedure. This approach assures uniqueness among concepts in the general and local KGs.

escape_special_chars(self, string_to_escape) deals with the tedious work of escaping the special characters that may appear in a user utterance. When using regular expressions, some characters may have special meanings to the interpreter leading to unwanted matches. Therefore, they must be escaped using the backslash sign. It is such an important task because all procedures base themselves on firstly searching for the right instances to be manipulated.

checkParams(self, params, state) checks the compulsoriness of the parameters associated with the type of the currently discussed instance. Before being inserted, an instance needs to have at least the mandatory parameters. Therefore, the function gets the state of the instance (that contains the mentioned parameters) and checks the inclusion of them. It returns the action to be executed by the TOD system.

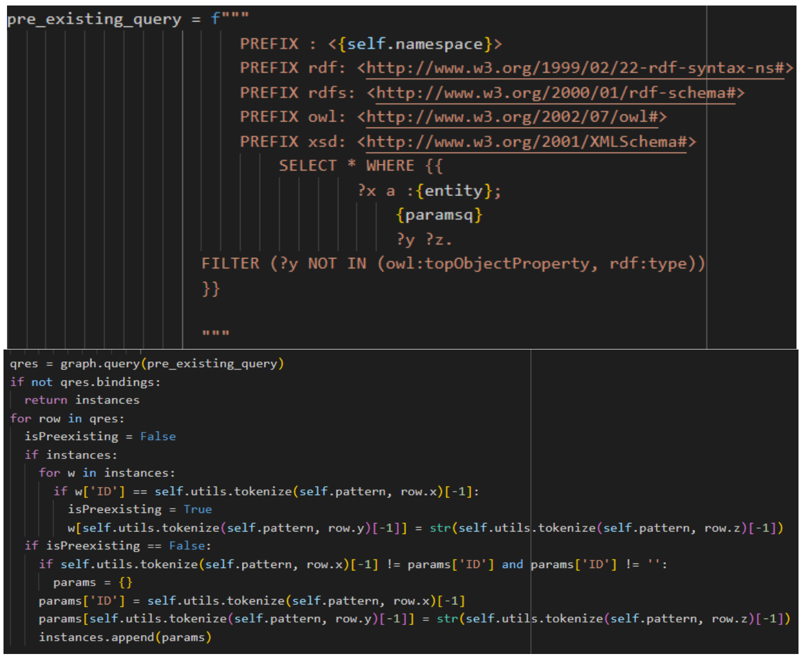
getInstanceOrEntity(self, tokens, intent_keywords, known_entities) executes a crucial task for the TOD system by getting the subject of discussion. Each user utterance may contain references to either an instance or entity type that the TODS has to be aware of. When invoked, this function loops through all the tokens of the user utterance, checks the intent of the phrase, and returns the subject of the discussion, if any.

getParamsValues(self, tokens, intent, params, slots, general_existence_words) identifies the values of the mentioned parameters in the user utterance. Using keyword detection and the active procedure, each parameter that is directly mentioned by name is retrieved from the text, together with the desired value. It plays a key role for TODS, which needs to be aware of the latest information provided by the user.

1.1.2 Query Class

To retrieve information from a KB, one needs to send queries and process the results. Query class holds together all the important interrogations sent to the general and local KG by any of the three components that form the M2M system. Each query follows the same structure: define its text, execute it, and return the response. Some do pre-processing of the results before sending them to the caller. Interrogations are defined for tasks, such as: getting the parameters associated with an entity type, checking the existence of a specific instance with or without parameters, returning the state of the currently discussed instance, verifying if an instance is linked to another one, getting information about a class or procedure, and counting the number of instances per each class.

One of the most used queries by the M2M system checks the pre-existence of an instance. The first part of Figure 7 depicts the body of the interrogation. Using the SPARQL language, it starts by mentioning the prefixes relevant for identifying a certain concept. One prefix is introduced dynamically, assuring that a query can adapt to new ontologies. Next, the main part is represented by a SELECT query, that retrieves all the instances of a specific type (*entity* argument) with (*paramsq* argument) or without (*?x ?z* syntax) certain relationships. The second part allows for a more particular or general verification. The last bit of the function pre-processes the output after querying the desired KG. It loops through each result (*row* variable), gets the necessary information, and stores it in a local dictionary. All returned instances are stored in different dictionaries identified by their associated ID. Each dictionary is inserted in a list, which is the final returned result. We only keep the final part of an ID, therefore we tokenize each output and save the last token. All functions have the same format, some doing more or less pre-processing.



```
pre_existing_query = f"""
          PREFIX : <{self.namespace}>
          PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
          PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
          PREFIX owl: <http://www.w3.org/2002/07/owl#>
          PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
              SELECT * WHERE {{
                      ?x a :{entity};
                          {paramsq}
                          ?y ?z.
      FILTER (?y NOT IN (owl:topObjectProperty, rdf:type))
      }}

      """
qres = graph.query(pre_existing_query)
if not qres.bindings:
  return instances
for row in qres:
  isPreexisting = False
  if instances:
    for w in instances:
      if w['ID'] == self.utils.tokenize(self.pattern, row.x)[-1]:
        isPreexisting = True
        w[self.utils.tokenize(self.pattern, row.y)[-1]] = str(self.utils.tokenize(self.pattern, row.z)[-1])
  if isPreexisting == False:
    if self.utils.tokenize(self.pattern, row.x)[-1] != params['ID'] and params['ID'] != '':
      params = {}
    params['ID'] = self.utils.tokenize(self.pattern, row.x)[-1]
    params[self.utils.tokenize(self.pattern, row.y)[-1]] = str(self.utils.tokenize(self.pattern, row.z)[-1])
    instances.append(params)
```

Figure 7. The structure of the pre-existing query.

1.1.3 Adding Parameter's Values

In both the prompt generator and user simulator, it is mentioned that values are added to parameters. The procedure is similar in both cases, but some differences are highlighted.

In the form of a function, the addition starts by gathering the parameters specific to the provided type. Then, it creates a list of possible parameters to which it can add values. In the case of the prompt generator, all parameters are available, while for the user simulator only the ones which were not mentioned in the discussion. The next step generates a sub-list, to simulate the situation where not all parameters are provided from the first user utterance. Permutations are used to produce non-identical lists of parameters each time the function is called. In case no parameters are available, the function stops. Otherwise, it starts generating values for either literal or concept parameters.

Literal values can be built in two ways:

- 70% of the time the value is exactly equal to one existing in the KG
- 30% of the time value is formulated such that it suggests a similarity to existing values from the knowledge graph ('something like *value'*). The value can be either the full word or a slice of it (with the minimum length being one). For example, if we have the word 'Python' as a value, a slice of length 2 would be 'Py'.

Concept values might refer to existing instances in the KG, and have four modes of being enunciated:

- 48% probability of referring by the value available at the hasName parameter
- 44% probability of formulating using general existence terms, either:
  - by referring to a known parameter, ‘someone with parameter value’
  - by referring to an unknown parameter, ‘someone with anything value’.
- 8% probability to set an ID for the value.

The general existence terms can vary, retrieving a value from the list: 'something',' anything', 'someone', 'somebody', 'anybody', 'anyone'. Some terms can be associated either with humans or things. Therefore, the addition of values is designed to semantically match the type of parameter. This is done by adding a label to each concept in the ontology, specifying the category to which they adhere.

A value is extracted from a pool of words that is mapped under the form of a text file. A pool is named by combining either the type of entity or instance with the name of the parameter, or the type of parameter and its name. The first case is used for literal values, while the second one refers to the type of concept associated with the parameter.

The prompt generator avoids adding concept values for the Insert procedure, conditioned by the spawning process of the user simulator. More exactly, spawning a new thread is related to the non-existence of a parameter’s value, thus the simulator needs to know the type of instance that will be inserted in the graph. It cannot know it directly from a prompt, but only from its internal adding parameter values function. When the user simulator adds values for parameters, it keeps track of the concept values that were inserted for each instance, to either spawn new sub-threads or confirm future insertion of them.

A value for a parameter is extracted from a sample pool. Currently, there are pools for all concepts in the ontology, but it will be a difficult task to scale for each new concept added to the ontology. This is going to be addressed in future versions of the system.

There are 7 pools available, 6 related to ontology and a special pool, as follows:

- *Projectname*, with 162 possible names for projects, like BestApp, Net-App, etc.
- *Projectclass*, having 20 different class names, for example, C++, Python, etc.
- *Projectcode*, holding 103 3-character long codes, including YM5, BR4, 123, etc.
- *Employeename*, where 1476 names are available, such as Ionut, John, Omar. etc.
- *Employeerole*, with 101 roles, like lawyer, analyst, CEO. etc.
- *Statusname*, containing 8 possible names, such as done, unfinished. etc.

- Special pool, *wrongentities*, having 11 concepts that are not included in the ontology.

4. Results

To test the capabilities of our system, we run experiments on datasets of size 1000, over ten times. The size is arbitrarily chosen, as it can be set to as much as wanted, but it is comparable with other important datasets, such as DSTC2 [31], SFX [32], WOZ2 [10], or Frames [21]. Table 1 shows the comparison between our dataset and other corpuses, according to [10]. Due to the fact that we only keep the user's utterances, some metrics could not be calculated, therefore they were removed. Compared to the other datasets, our corpus contains 1000 dialogues, the highest among all. Also, the system's capacity of generating sub-words from the sample pools increases the number of different values to 6915, more than any other dataset. The average number of turns per dialogue is relatively low compared to the other corpuses, highlighting the lack of human intervention. Overall, the metrics reveal that the M2M system can generate datasets comparable to the SOTA ones.

Each round of validation starts from the given general KB which contains eight default instances and produces 1000 scenarios. After the round ends, the system resets the general KB to the default state and re-generates scenarios.

Table 1. Comparison of our dataset to other similar datasets, according to [10].

Metric	*DSTC2*	*SFX*	*WOZ 2.0*	*FRAMES*	*KVRET*	*M2M*	*MultiWOZ*	*Dialogue Simulator (our work)*
# Dialogues	1,612	1,006	600	1,369	2,425	1,500	8,438	**10,000**
Total # turns	23,354	12,396	4,472	19,986	12,732	14,796	**113,556**	80,799
Avg. turns per dialogue	14.49	12.32	7.45	**14.6**	5.25	9.86	13.13	7.135
# Slots	8	14	4	**61**	13	14	24	25
# Values	212	1847	99	3871	1363	138	4510	**6915**

In our research, the number of scenarios is fixed at 1000. The time needed to generate them is, on average, 21 minutes. It is a reasonable result, as it has low costs in terms of time, human and financial resources. Table 1 presents the outcome for each round of validation.

Table 2. Execution time (in minutes) for each validation round.

Validation round	*Execution time (minutes)*
1	21
2	21
3	20
4	19
5	17
6	20
7	23
8	19
9	21
10	22

1.1.1 Average Turns per Procedure

The four possible procedures (Select, Update, Delete, and Insert) differ in terms of complexity from one another. The first three can have either one or two tasks in the generated prompt. Therefore, we provide both an extended and summarized analysis of the average turns per procedure.

As each prompt is designed to have at least three tasks, any procedure has a minimum of four turns. Therefore, the average will always be equal to or greater than four for select1, update1, and delete1 (category procedure1) and six for select2, update2, and delete2 (category procedure2). In Figure 8, each procedure is divided into sub-categories. Figure 9 summarizes the results of the main procedures.

As expected, select1 and 2 have a value closer to the minimum turns' value (4.24 > 4 and 6.43 > 6) as it has only four possible routes of discussion, out of which only one continues the task. Delete has five expected system actions, one more than select, leading to higher average values (4.34 > 4 and 6.72 > 6). Update, with its nine system actions, manages to extend the average turns up to one per scenario, which is better than Select or Delete. This means that the M2M system makes use of all the built-in routes it knows. Insert is the leading procedure in terms of results. With an average of 11.7 turns per scenario, with 7.7 more than the expected minimum, it proves that the M2M system spawns sub-threads of discussion according to the given probabilities and can carry on a conversation toward solving the user's tasks.

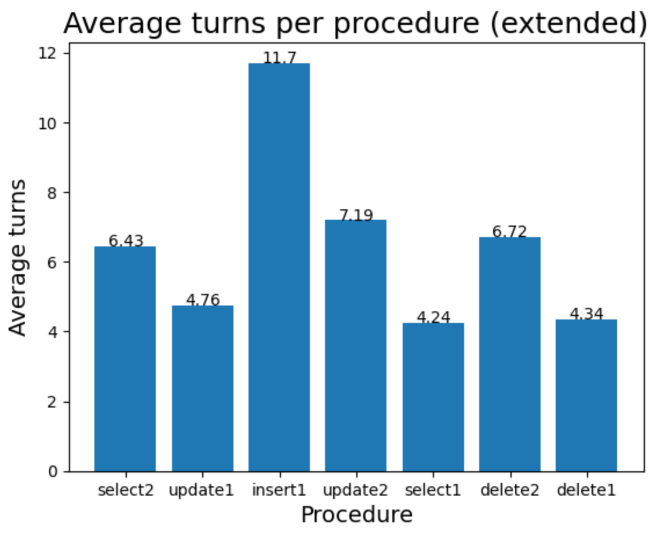


Figure 8. Average turns per procedure, extended version.

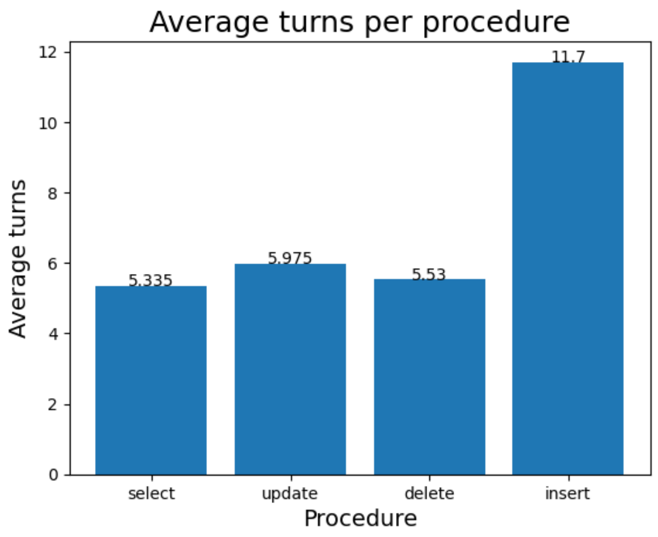


Figure 9. Average turns per procedure, summarized version.

1.1.2 Maximum Number of Turns per Procedure

Two main factors can increase the number of turns: the probability paths defined inside the user simulator and the number of instances available in the general KB.

The probability paths are defined in the development stage; therefore, they are fixed. This results in preliminary assumptions about the evolution of turns' number, such that if the number of possible system actions is lower, so is the probability to continue the discussion. However, the size of the general KB fluctuates throughout the generation of scenarios, thus influencing the final number of turns. For example, a discussion thread is prone to finishing earlier if the queried instances do not exist in the general KB.

Table 2 proves our expectations, as Select and Delete procedures (each having four, respectively five system actions to react to) have the maximum number of turns around 12, between six and seven more than the average. Update is a more complex procedure, that has to first search for the existence of the desired instances to be updated, check the correctness of the provided values, and update the instances if everything is fine. Therefore, it may take longer to reach the goal, with the maximum registered number of turns being 28. Lastly, Insert is the only procedure that can spawn sub-threads of discussion, leading to a scenario of 197 turns. Compared to the 11.7 average, it can be labeled as an outlier, which happened in the first 20% generated scenarios, when the general KB does not have many instances, thus the need to spawn sub-threads to insert each requested instance, but it showcases the endurance of the M2M system to accomplish the given task. Even though is less likely to be considered a real-life scenario, it adapts the system to unexpected behavior from users.

Table 3. The maximum number of turns per procedure, extended version.

Procedure (extended)	*Maximum turns per scenario*
select1	11
select2	12
update1	28
update2	22
delete1	8
delete2	11
insert1	197

5. Conclusions

Our work achieved the development of a dialogue simulator, capable of generating an arbitrary amount of scenarios, thus reducing the resources needed to obtain a custom dataset for training the Deep Learning components of a TOD system, that may be of interest to a business organization.

The TODS component successfully integrated a given ontology as its knowledge and communicated via natural language with the end user. The Prompt Generator and User Simulator worked together towards creating different scenarios and transforming them into natural language user utterances. All three components aligned themselves to grow the knowledge base of a target company and to prepare future datasets with annotated conversations regarding the business procedures taking place within the firm.

Having a rule-based design, it is limited by the engineered setup of the user and system bots. Besides, it requires a list of possible intents and templates for natural language responses, thus limiting its knowledge and behavior.

Future work will focus on training Deep Learning components with the datasets obtained by using the described dialogue simulator, followed by integrating them into the general architecture. This creates a recursive loop that can provide better datasets and increase the performance of the overall system. Finally, after all components have incorporated Deep Learning models, a standalone TOD system will be ready to be deployed.

Acknowledgements

This work received support through the project: Integrated system for automating business processes using artificial intelligence, POC/163/1/3/121075 - a Project Co-financed by the European Regional Development Fund (ERDF) through the Competitiveness Operational Programme 2014-2020.

I'm gratefully to Babes-Bolyai University for awarding me the scholarship for scientific performance during 2022-2023 academic year.

Experiments were performed on the HPC infrastructure of Babes-Bolyai University, developed with the support of the European Regional Development Fund of European Union under projects SmartCloudDSS (cod SMIS 124155) and Madecip (cod SMIS 48801/1862).

Bibliography

1. Adiwardana, D., Luong, M.-T., So, D.R., Hall, J., Fiedel, N., Thoppilan, R., Yang, Z., Kulshreshtha, A., Nemade, G., Lu, Y., Quoc V.L. (2020), Towards a Human-like Open-Domain Chatbot. Available at: https://arxiv.org/abs/2001.09977

2. Zhang, Z., Ryuichi, T., Qi, Z., MinLie, H. and XiaoYan, Z. (2020), Recent Advances and Challenges in Task-Oriented Dialog Systems, *Science China Technological Sciences 63 (10): 2011-2027*.

3. Jurafsky, D. and Martin, J. H. (2021), *Speech and Language Processing*, 3rd draft, chapter 24, Stanford University.

4. Su, Y., Shu, L., Mansimov, E., Gupta, A., Cai, D., Lai, Y.-A. and Zhang, Y. (2022), Multi-task pre-trained for plug-and-play task-oriented dialogue system, *Proceedings of the 60th annual meeting of the Association for Computational Linguistics (ACL 2022), vol 1: (long papers)*, pp. 4661-4676.

5. Chen, Z., Chen, L., Chen, B., Qin, L., Liu, Y., Zhu, S., Lou, J.-G. and Yu, K. (2022), UniDU: towards a unified generative dialogue understanding framework, *Proceedings of the SIGdial 2022 Conference*, pp. 442–455.

6. Wu, C.-S., Hoi, S., Socher, R. and Xiong, C. (2020), TOD-BERT: Pre-trained Natural Language Understanding for Task-oriented dialogue, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 917-929.

7. Li, Z., Wang, H., Albalak, A., Yang, Y., Qian, J., Li, S. and Yan, X. (2021), Making Something out of Nothing: Building Robust Task-oriented Dialogue Systems from Scratch, *1st Proceedings of Alexa Prize TaskBot (Alexa Prize 2021)*.

8. Andreas, J. et al (2020), Task-oriented dialogue as a dataflow synthesis, *Transactions of the Association for Computational Linguistics*, vol. 8, pp. 556-571.

9. Yan, Z., Duan, N., Chen, P., Zhou, M., Zhou, J. and Li, Z. (2017), Building Task-Oriented Dialogue Systems for Online Shopping, *31st AAAI Conference on Artificial Intelligence*, pp. 4618-4625.

10. Wen, T.-S., Vandyke, D., Mrksic, N., Gasic, M., Rojas-Barahona, L.M., Su, P.-H., Ultes, S. and Young, S. (2017), A network-based end-to-end trainable task oriented dialogue system, *15th Intl Conference of the European Chapter of ACL*, vol. 1, pp. 438-449.

11. Chen, Q., Zhuo, Z., Wang, W. (2019), BERT for Joint Intent Classification and Slot Filling. Available at: https://arxiv.org/abs/1902.10909

12. Devlin, J., Chang, M.W., Lee, K., Toutanova, K. (2019), BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human*

Language Technologies, vol. 1., pp. 4171-4186

13. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W. and Liu, P. J. (2020), Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer, *Journal of Machine Learning Research 21*.

14. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., *et al*. (2020), Language Models are Few-Shot Learners, *NIPS'20: Proceedings of the 34th International Conference on Neural Information Processing SystemsDecember 2020*, Article No.: 159, Pages 1877–1901.

15. Zhang, Z., Zhang, Z., Chen, H. and Zhang, Z. (2019), A Joint Learning Framework with BERT for Spoken Language Understanding, *IEEE Access 7,8907842*, pp. 168849-168858

16. Coucke, A., Saade, A., Ball, A., et al. (2018), Snips Voice Platform: an embedded Spoken Language Understanding system for private-by-design voice interfaces. Available at http://arxiv.org/abs/1805.10190

17. Tur, G., Hakkani-Tur, D. and Heck, L. (2010), What is left to be understood in ATIS?, 2*010 IEEE Spoken Language Technology Workshop*.

18. Casanueva, I., Temčinas, T., Gerz, D., Henderson, M. And Vulić, I. (2020), Efficient Intent Detection with Dual Sentence Encoders, *Proceedings of the 2nd Workshop on Natural Language Processing for Conversational AI*, pages 38–45.

19. Coope, S., Farghly, T., Gerz, D., Vulić, I. and Henderson, M. (2020), Span-ConveRT: Few-shot Span Extraction for Dialog with Pretrained Conversational Representations, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 107–121.

20. Budzianowski, P., Wen, T.S., Tseng, B.H., Casanueva, I., Ultes, S., Ramadan, O. And Gašić, M. (2018), MultiWOZ – A large scale multi-domain Wizard-of-oz dataset for task-oriented dialogue modelling, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language* Processing, pp. 5016-5026.

21. El Asri, L., Schulz, H., Sharma, S., Zumer, J., Harris, J., Fine, E., Mehrotra, R. and Suleman, K. (2017), Frames: a corpus for adding memory to goal-oriented dialogue systems, *Proceedings of the 18th Annual SIGdial Meeting on Discourse and Dialogue*, pages 207–219.

22. Byrne, B., Krishnamoorthi, K., Sankar, C., Neelakantan, A., Goodrich, B., Duckworth, D., Yavuz, S., Dubey, A., Kim K.-Y. and Cedilnik, A. (2019), Taskmaster-1: Toward a Realistic and Diverse Dialog Dataset, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4516–4525.

23. Rastogi, A., Zang, X., Sunkara, S. (2020), Towards scalable multi-domain conversational agents: The schema-guided dialogue dataset, *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 8689-8696.

24. Li, X., Wang, Y., Sun, S., Panda, S., Liu J. and Gao, J. (2018), Microsoft Dialogue

Challenge: Building End-to-End Task-Completion Dialogue System", *2018 IEEE Spoken Language Technology Workshop (SLT)*.

25. Mancini, M. (2019), Bootstrapping Dialog data, annotations and models. *University of Trento*, Master Thesis. Available at: https://github.com/marcomanciniunitn/Master-Thesis-Project/blob/master/Thesis.pdf

26. Shah, P., Hakkani-Tür, D.Z., Tür, G., Rastogi, A., Bapna, A., Kennard, N.N. and Heck, L. (2018). Building a Conversational Agent Overnight with Dialogue Self-Play. Available at: https://arxiv.org/abs/1801.04871

27. Tuan, Y., Beygi, S., Fazel-Zarandi, M., et al. (2022), Towards Large-Scale Interpretable Knowledge Graph Reasoning for Dialogue Systems, *Findings of the Association for Computational Linguistics: ACL 2022*, pp. 383–395.

28. Yang, S., Zhang, R. and Erfani, S. M. (2020), Graphdialog: Integrating graph knowledge into end-to-end task-oriented dialogue systems, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, pages 1878–1888.

29. Chaudhuri, D., Rony, M.R.A.H. and Lehmann, J. (2021), Grounding Dialogue Systems via Knowledge Graph Aware Decoding with Pre-trained Transformers, *ESWC 2021: The Semantic Web*, 12731, pp. 323–339.

30. Bordes, A., Boureau, Y-L. and Weston, J. (2017), Learning end-to-end goal-oriented dialog, *ICLR 2017*.

31. Henderson, M., Thomson, B. and Williams, J. (2014a), The second dialog state tracking challenge, *Proceedings of SIGdial*.

32. Gasic, M., Kim, D., Tsiakoulis, P., Breslin, C., Henderson, M., Szummer, M., Thomson, B. and Young, S. (2014), Incremental online adaptation of pomdp-based dialogue managers to extended domains, *Interspeech*.

33. V.I. Iga, G.C. Silaghi (2023), Ontology-based dialogue system for domain-specific knowledge acquisition, Proceedings of the 31st International Conference on Information Systems Development, Lisbon, Portugal, to appear

Annexes

Annex 1. The evolution of a conversation with multiple threads

Figure 10 depicts a discussion between a human user and the TOD system about the insertion of a Project instance. It leads to three different discussion threads, as described below.

At the very beginning, the user says "Insert a project...", which triggers the detection of the project concept from the general KG and the generation of a unique node in the local KB. Then, it gets the specific parameters of the discovered concept, starts an Insert procedure on the first thread, and continues the conversation toward collecting all the requested information for the project. If the user mentions an unknown entity (from the perspective of the general KG) the system defaults. In the same utterance, the user mentions values for some parameters and uses keyword detection to identify them (here code: 123, class: Python, name: TaskHelper, status: unfinished, manager: John). The system defaults if there is no active ID.

At turn 3, the user changes his mind and decides to remove a parameter by mentioning its value (here, remove John). The system supports two types of removal: by name of the parameter or by value. If the user mentions a wrong name or value, the system defaults.

At turn 4, the user decides to initiate a new procedure by saying “Please insert a new one”. The system does its routine check and discovers that no new concept was mentioned, which triggers the creation of a procedure with the old mentioned type (stored locally, here Project). The now active ID is placed in a Last-In-First-Out (LIFO) style list called “predecessor” and a new ID is generated and set as active, which starts the second thread. The list is the main component of the multiple threads architecture, alongside the local graph. In it, all the procedures that were once active, but never finished (either by inserting it in the general KB or stopped by the user) are saved for later reference.

In turns 5-6, the conversation continues toward building the active procedure (a project instance). At turn 6, the system warns the user that they referred to an Employee instance that does not exist in the graph (he wanted to assign Marc as the manager, but there is no Marc in the graph). Therefore, the user needs to insert an employee called Marc, and then proceed with the project insertion.

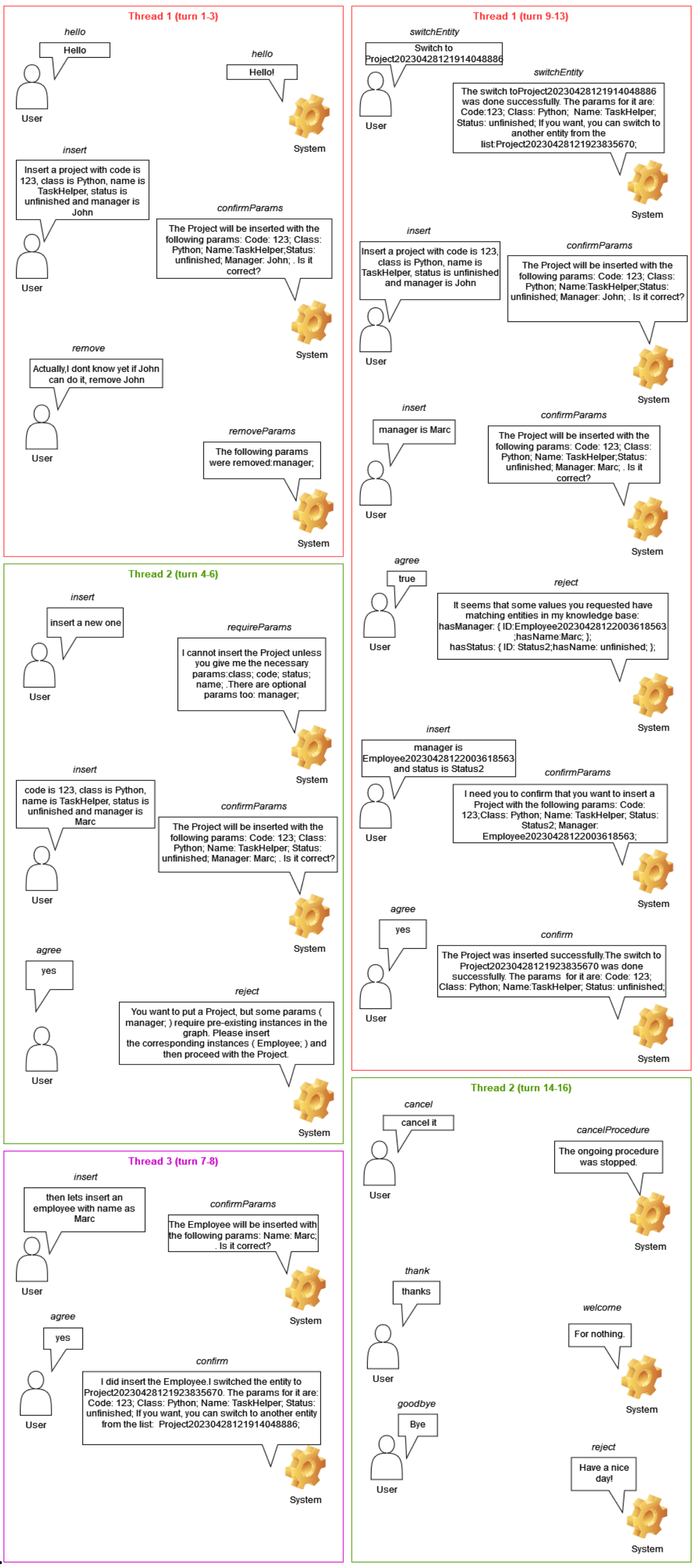
In turn 7, the user complies with the system request and initiates another procedure. Eventually, the third thread is created, while the system places the current active ID in the predecessor list and generates a new procedure according to the new type given by the user (here, Employee). As the user gave all the mandatory parameters (here, only the name), the system directly asks for confirmation of insertion.

In turn 8 the user confirms the correctness of the Employee instance and the system successfully inserts it in the general KB, which also ends the third thread. At the same time, it checks for instances in the predecessor list. If there is any, the system pops the last one inserted (LIFO approach) and gets its state from the local graph, to show it to the user. This switches the conversation to the second thread. Also, it prompts that the user can switch to another active instance if there is any.

In turn 9, the user decides to switch to the first inserted instance (first thread). The system places the current active ID back in the predecessor list and sets the mentioned ID as active. It similarly loads its state as in turn 8.

After a few turns of discussing the current active instance, in turn 12 the system prompts the user that it found instances matching their filters in the general KB and they need to choose the correct IDs for each parameter. When a parameter requests a pre-existing instance in the general KB, the system always checks its existence in two ways: by ID or by a parameter value. The system can directly check the existence of an instance by its ID. Since this is not an often case, it allows checking by parameters' values. The system gets the mentioned value (here, Marc and unfinished) and checks each parameter of every instance of the appropriate type. If values are found, they are prompt to the user.

Turn 13-14 builds the final form of the project instance. After successful insertion of it in the general KB, the system switches to the last active instance from the predecessor list. This sets the second thread active again. In turn 15, the user cancels it, which triggers the removal of all associated information from the local graph. If no instances are remaining in the predecessor list, the system does not show the switch option. Turn 15-17 conclude the conversation. The system successfully managed three active threads, with two insertions and a canceled procedure.


Thread 1 (turn 1-3)

User (*hello*): Hello

System (*hello*): Hello!

User (*insert*): Insert a project with code is 123, class is Python, name is TaskHelper, status is unfinished and manager is John

System (*confirmParams*): The Project will be inserted with the following params: Code: 123; Class: Python; Name:TaskHelper;Status: unfinished; Manager: John; . Is it correct?

User (*remove*): Actually,I dont know yet if John can do it, remove John

System (*removeParams*): The following params were removed:manager;

Thread 2 (turn 4-6)

User (*insert*): insert a new one

System (*requireParams*): I cannot insert the Project unless you give me the necessary params:class; code; status; name; .There are optional params too: manager;

User (*insert*): code is 123, class is Python, name is TaskHelper, status is unfinished and manager is Marc

System (*confirmParams*): The Project will be inserted with the following params: Code: 123; Class: Python; Name: TaskHelper; Status: unfinished; Manager: Marc; . Is it correct?

User (*agree*): yes

System (*reject*): You want to put a Project, but some params (manager;) require pre-existing instances in the graph. Please insert the corresponding instances (Employee;) and then proceed with the Project.

Thread 3 (turn 7-8)

User (*insert*): then lets insert an employee with name as Marc

System (*confirmParams*): The Employee will be inserted with the following params: Name: Marc; . Is it correct?

User (*agree*): yes

System (*confirm*): I did insert the Employee.I switched the entity to Project20230428121923835670. The params for it are: Code: 123; Class: Python; Name: TaskHelper; Status: unfinished; If you want, you can switch to another entity from the list: Project20230428121914048886;

Thread 1 (turn 9-13)

User (*switchEntity*): Switch to Project20230428121914048886

System (*switchEntity*): The switch toProject20230428121914048886 was done successfully. The params for it are: Code:123; Class: Python; Name: TaskHelper; Status: unfinished; If you want, you can switch to another entity from the list:Project20230428121923835670;

User (*insert*): Insert a project with code is 123, class is Python, name is TaskHelper, status is unfinished and manager is John

System (*confirmParams*): The Project will be inserted with the following params: Code: 123; Class: Python; Name:TaskHelper;Status: unfinished; Manager: John; . Is it correct?

User (*insert*): manager is Marc

System (*confirmParams*): The Project will be inserted with the following params: Code: 123; Class: Python; Name: TaskHelper;Status: unfinished; Manager: Marc; . Is it correct?

User (*agree*): true

System (*reject*): It seems that some values you requested have matching entities in my knowledge base: hasManager: { ID:Employee20230428122003618563 ;hasName:Marc; }; hasStatus: { ID: Status2;hasName: unfinished; };

User (*insert*): manager is Employee20230428122003618563 and status is Status2

System (*confirmParams*): I need you to confirm that you want to insert a Project with the following params: Code: 123;Class: Python; Name: TaskHelper; Status: Status2; Manager: Employee20230428122003618563;

User (*agree*): yes

System (*confirm*): The Project was inserted successfully.The switch to Project20230428121923835670 was done successfully. The params for it are: Code: 123; Class: Python; Name:TaskHelper; Status: unfinished;

Thread 2 (turn 14-16)

User (*cancel*): cancel it

System (*cancelProcedure*): The ongoing procedure was stopped.

User (*thank*): thanks

System (*welcome*): For nothing.

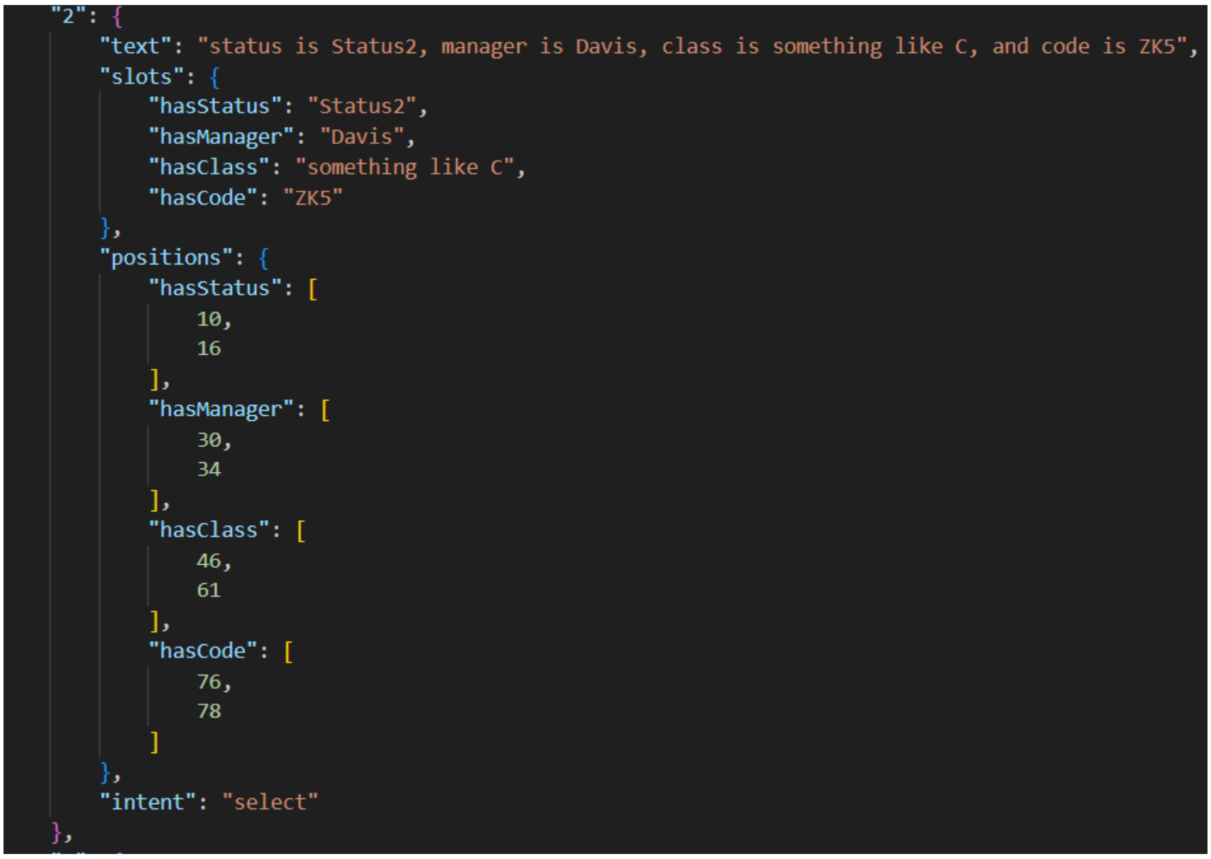
User (*goodbye*): Bye

System (*reject*): Have a nice day!

Figure 10. Conversation example with multiple discussion threads.

Annex 2. The structure of a dataset labeled for intent detection and slot filling

The figure shows one annotated user utterance, where each detected slot is saved in a dictionary with the key as the parameter's name and the value as its corresponding text sequence. All slots are identified using their starting and ending indexes in the text. Finally, the detected intent is inserted under the 'intent' key. All user utterances respect this structure.



```
"2": {
    "text": "status is Status2, manager is Davis, class is something like C, and code is ZK5",
    "slots": {
        "hasStatus": "Status2",
        "hasManager": "Davis",
        "hasClass": "something like C",
        "hasCode": "ZK5"
    },
    "positions": {
        "hasStatus": [
            10,
            16
        ],
        "hasManager": [
            30,
            34
        ],
        "hasClass": [
            46,
            61
        ],
        "hasCode": [
            76,
            78
        ]
    },
    "intent": "select"
},
```

Figure 11. The structure of a user utterance labeled for the NLU task.