FASTDial: Abstracting Dialogue Policies for Fast Development of Task Oriented Agents

Serra Sinem Tekiroğlu and **Bernardo Magnini** and **Marco Guerini**
Fondazione Bruno Kessler, Via Sommarive 18, Povo, Trento, Italy
tekiroglu@fbk.eu, magnini@fbk.eu, guerini@fbk.eu

Abstract

We present a novel abstraction framework called FASTDial for designing task oriented dialogue agents, built on top of the OpenDial toolkit. This framework is meant to facilitate prototyping and development of dialogue systems from scratch also by non tech savvy, especially when limited training data is available. To this end, we use a generic and simple frame-slots data-structure with pre-defined dialogue policies that allows for fast design and implementation at the price of some flexibility reduction. Moreover, it allows for minimizing programming effort and domain expert training time, by hiding away many implementation details.

1 Introduction

In recent years, there has been an increasing demand for a new generation of conversational systems that are able to naturally interact and assist humans in a number of scenarios, including - but not limited to - virtual coaches, personal assistants and automatic help desks. However, when dealing with applications or commercial scenarios, technological complexity should be abstracted away since domain knowledge is often held by non tech savvy. Moreover, systems should be 'transparent' to easily allow for modification or scaling when needed, such as error fixing or new intents/objects/requirements integration.

To this end, several solutions have appeared on the market. On one side, there are open source tools/frameworks, such as OpenDial (Lison and Kennington, 2016), PyDial (Ultes et al., 2017) and DeepPavlov (Burtsev et al., 2018) that are very flexible and allow many integrations. While these tools are designed with the target of computer scientists in mind, they would still need domain expertise to design proper dialogues. On the other side of the spectrum, several commercial tools aim at hiding dialogue implementation complexity, for example by using intuitive graphical interfaces to help also non technical experts build their own dialogues. This comes to the price of loosing some flexibility, integration capabilities, and control over the system.

However, constructing flexible multi-intent dialogue agents while keeping the implementation complexity minimal is not a trivial task both on commercial and open source tools. Considering that there are several domains requiring such dialogue systems (e.g. banking, e-commerce etc.), we aim at providing a framework that is easy to use but at the same time is still as flexible as possible.

To this end, we tried to merge the best of both worlds (commercial and open source tools) by designing and implementing a generalization architecture on top of OpenDial. OpenDial is a Java-based, domain-independent framework for developing probabilistic rule-based dialogue systems. While keeping the rule-based approach of OpenDial, our architecture, named FASTDial, abstracts away dialogue policies in a generic dialogue model that drastically reduces design effort and code complexity. This generic dialogue model starts with an intent recognition phase that is user initiative. Once the intent is recognized the interaction is converted to system initiative for filling the required slots. Still, the user is given an active part by allowing universal interruptions such as calling help or canceling the task (Jurafsky and Martin, 2017). In our view, most task oriented domains can be easily adapted to this schema. This allows for speeding up the prototyping of complex multi-intent dialogue scenarios. It allows easy integration of new languages and new intents, by prioritizing efficiency and extensibility to facilitate developing dialogue systems from scratch and with limited training data available. Scalabil-

Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pages 75–80
Florence, Italy, July 28 - August 2, 2019.

ity can be quickly obtained also by non-experts since the technical implementation of the dialogue flow is abstracted away. Therefore, instead of focusing on a graphical interface approach, we abstracted the dialogue policy in order to make dialogue building simpler:

- non-experts can easily be trained to write dialogues - or better - to provide the information needed to automatically establish a dialogue.
- new intents can be quickly added.
- dialogues can be quickly and easily ported to new languages.
- by using API interface to separate dialogue from actual data we can increase the modularity in the applications.

However, generalizing the dialogue flow into certain logical patterns brings along the restriction in the dialogue policy flexibility such that the agent can only handle informable slot types and a single slot per turn.

The remainder of the paper is structured as follow: Section 2 presents the architecture of our system with its main components, while Section 3 presents our running example in the banking scenario. Finally, in Section 4 and 5 we evaluate our approach and discuss future developments.

2 Architecture

In the FASTDial architecture[1], the main conversational ability of the agent is encoded in a generalized dialogue policy mechanism that is fed with a *frame-slots data-structure* (FSDS). By abstracting the policy away, we manage a systematic way to load new intents (tasks/goals) to the agent represented only as slots, data_types, api calls, and pre-defined system utterances to produce relevant dialogues with the user.

In this scenario, each intent relevant to a given domain (e.g. 'make a money transfer', 'check account balance', 'block credit card') is represented by one FSDS data structure. After recognizing the user intent, the agent loads the corresponding FSDS from an external resource (i.e. json files) and interactively fills the slots by asking related questions to the user, making sure that all the constraints associated with each slot are fulfilled. By design, the agent performs single-intent at a time and after each execution, the user can request a new intent. An intent can be categorized as either a query intent or an action intent. Regarding the banking domain, requesting information on the *Account Balance* is a query intent, while making a *Money Transfer* is categorized as an action intent.

The architecture has been designed to be handy especially in the application scenarios where many user intents must be implemented in a very short time and the training dialogue data is really scarce. The main framework is designed as a module on top of OpenDial toolkit. The architecture of the dialogue agent consists of 5 components; namely, Dialogue Manager, Integration Interface, Natural Language Understanding, Natural Language Generation, and Intent Manager.

2.1 Dialogue Manager Module

The dialogue manager is responsible of the inter-module communication and controlling the dialogue flow through dialogue states. In FASTDial, once the user intent is identified, the dialogue is led by the system. With respect to the slot order in the FSDS of the intent, the system asks the slot filling questions and processes the respective user utterance to retrieve the slot values. After the slot value is extracted, the DM module sends it to the API module for validation. If the value is valid, the DM module either activates the next slot state or finalizes the dialogue if all the mandatory slots are filled. If the value is not valid, the DM reformulates the machine utterance with the corresponding validation error and asks the slot filling question again. In Figure 1, we give an example dialogue demonstrating the aforementioned DM functioning.

Although FASTDial is designed to be system initiative during slot filling, slots can be expressed by the user at the intent utterance state, which is the initial stage of the conversation, (e.g. "I would like to transfer *AMOUNT* euros to *NAME*"). For this reason all slots described in the FSDS are also searched in the intent utterance. Consequently, as well as detecting the *Money Transfer* intent in the example above, the system would capture the values for the *transfer destination* and the *amount* slots. The system validates the detected values and if no error occurs the system skips the corresponding slot filling questions, i.e. *SKIPPED_TURN* in Figure 1.

[1]FASTDial code can be downloaded at the following link: https://github.com/serrasinem/FASTDial

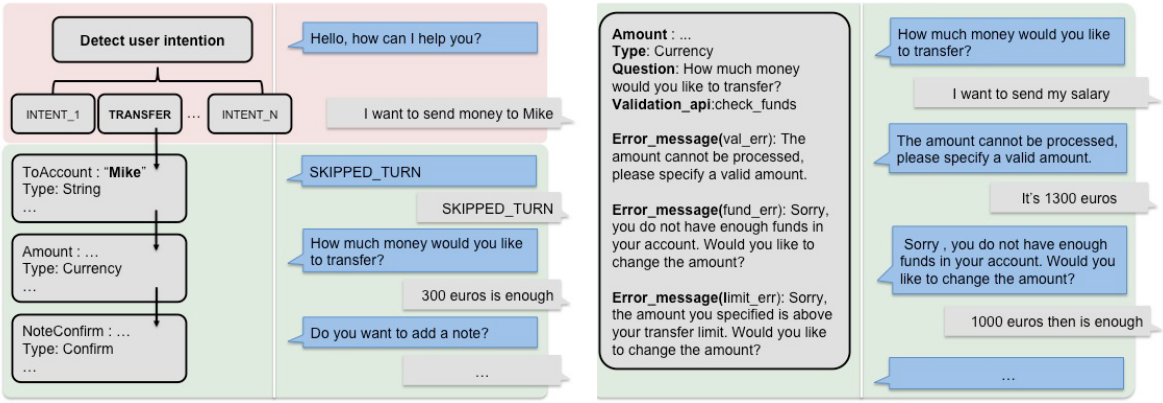

Figure 1: Example of dialogue with FASTDial in a multi-intent banking scenario. On the left: Upper section is intent recognition (user initiative). Once the intent has been recognized the system takes the lead and ask one slot at a time (lower section). On the right: a complex sub-dialogue due to validation errors for the amount slot.

FASTDial reduces the dialogue management complexity and provides a set of predefined dialogue policies, which most dialogue scenarios can be fitted into. However, the reduction of the dialogue flow complexity brings some restrictions to the flexibility of the dialogue management. We provide a list of restrictions and their alternatives in FASTDial dialogue flow as follows:

- system initiative dialogue flow: once the user intent is identified, the system tries to fill all necessary slots and in case of an unexpected user utterance, it changes the state to either repeating the question or finalizing the dialogue. This would prevent the system answering the mid-dialogue user questions, i.e. requestable slots (Henderson et al., 2013). As an example, in the *Money Transfer* intent, the user may ask about her account balance before specifying the amount of money she wants to send. The DM cannot change the intent to *Account Balance* query in this scenario.

- one slot per turn: a slot turn can only be for a single slot value. Multiple information must be asked in different questions as different slots. For instance, in a food ordering scenario, food type and number of orders should be defined as different slot turns. Still, both slots can be identified in the intent sentence if specified by the user.

2.2 Integration Interface

The dialogue agent can be used as an external tool to any application through its web service interface. Similar to the dialogue bAbI tasks (Bordes et al., 2016), the interface creates either a machine response or a functional call to its client. The responses of these functional calls, in a "variable:value format", are directed to the NLG module to transform the response values into suitable natural text shown to the user. In addition, for demo purposes, the framework has a simple Telegram application that requires a knowledge base implementation as a backend. Unlike the web service interface, Telegram app only produces machine utterances and connects to the backend knowledge base when necessary.

2.3 Natural Language Understanding

The task of the Natural Language Understanding component is twofold: first to recognize the user intent and then fill the corresponding FSDS that has been loaded by the Intent Module and active at the moment. While understanding the intent type is implemented as a model interface, the slot recognition task is handled by each slot type separately. The intent identification model is generated during the initialization of the agent, by retrieving the intent keys from all registered intents.

Each slot type has its own slot filling method and the type-specific slot constraints are employed to match the values from the user utterance. Therefore, the agent can employ different NLU approaches for different data types. Similar to the strategy in OpenDial, while a simple regular expression model is applied to retrieve numbers, a more complex deep learning recognition model can be applied to other slots. This approach allows integrating new slot filling models, such as Named Entity Recognition models (Louvan and

Magnini, 2018) or RNN (Mesnil et al., 2015), easily by overloading the matching function in the abstract class of the slot object.

2.4 Natural Language Generation

The machine utterances shown to the user throughout the dialogue are supplied by the Natural Language Generation module whose main responsibility is to select the correct machine response to lead the dialogue to successfully fulfill a user intent. In the startup of the agent, the NLG model is generated by using the intent descriptions retrieved from all the registered FSDSs and language-specific generic texts. API responses are also integrated into machine utterances whenever necessary. The machine utterances include the user greetings, the slot filling questions, confirmation requests, the output of the user intent executions, error messages (i.e. erroneous slot values, incomprehensible user utterances, and translation of the API responses with error codes).

2.5 Intent Manager

Intent Manager loads the intent frame (i.e. FSDS) on-the-fly when the user intent is detected by the NLU module. All possible FSDS, including their metadata and slot descriptions, are registered when the agent starts up. The intent frame descriptions reside in the FSDS Knowledge Base (KB), currently a simple folder containing json files of all registered intents. A new intent can be registered to the dialogue agent by adding it to the FSDS KB folder after the following properties are defined:

- name: defines the name of the intent.
- keys: defines the keywords that are necessary for understanding the user intent. It is a comma separated list of regular expressions. In addition, a more complex NLU function, such as any defined slot filling model, can be assigned to be called in this variable.
- confirmation: a boolean value determines if the execution of the intent requires a confirmation from the user. For instance, certain actions such as payments or, more generally, the tasks that require the user's complete awareness of the consequences should be finalized after the user's confirmation.
- confirmation question : if a confirmation is required for the execution of the intent, the confirmation question must be saved in this parameter.
- execution call: the final execution APICall. API formalism should be agreed upon with the middleware.
- success & error messages: the messages to be shown to the user when the execution of the intent is successful and when the execution of the intent is failed.
- slots: the 'ordered' list of slot objects that need to be filled to execute the intent.

A slot object can be in the type of String, List of Strings, Date, Time, Currency, Numeric, and Confirmation. Although we provide a pre-implemented list of possible slot types, the slot implementation has been designed as an abstract object that can be extended easily into new types of slots depending on the specific requirements of a new domain. For instance, a Named Entity slot that holds any given named entity can be implemented by employing a deep NER model. A slot object can be added to the slot list of an intent by defining the following variables.

- slot name: should be unique for intent.
- slot type: holds a slot type.
- constraint: type dependent. It determines the slot filling conditions necessary for the NLU.
- question: It holds the question to be asked to the user to fill the slot value.
- validation api: Each slot should be verified after receiving the value from the user in order to continue to the next slot, unless specified with the key "NoValidate". This variable holds the validation api function name to call after filling the slot value.
- error message: The error machine utterances and the error codes are defined in a list of "error_code:utterance" format.
- mandatory: This variable is a boolean value determining if the slot must be filled in order to execute the user intent.
- regex: We can define a list of regular expressions for extracting a slot value. Each regex

should contain "{slot}" string, which is the filler for the expected slot value, e.g. "my {slot} card". While "keys" parameter predefines a set of slot values to be exactly matched, "regex" parameter defines a set of patterns to extract the unknown slot values from the user utterance.

- dependency: The possible values that a slot can take sometimes depend on a previous slot. In this condition, dependency variable is required to hold a slot name which is defined prior to the current slot.

3 Banking Domain

Considering the data scarcity, Banking dialogues could be one of the hardest datasets to retrieve due to security requirements. Not only the real user interaction dialogues, but also the KB structure or the logical structure of a banking action would not completely and easily been shared. In this scenario, training a model with the actual banking data or dialogue samples would not be possible. Keeping these requirements and limitations in mind, we generated FSDSs for various banking tasks, such as *Money Transfer*, *Account History Search*, *Account Balance* query, *Card Limit* query, *Canceling a Transfer*, and *Blocking a Card*, from scratch with the help of a domain expert. In this domain, intent types and almost all slot values detected in the user utterances are validated through API calls and the dialogue flow can change accordingly. In Listing 1, we provide the json object of a sample FSDS registered to accomplish a *Money Transfer*. It is worthwhile to mention that to give the ability of establishing dialogues for the *Money Transfer* intent, all required information resides in this single json object. Additionally, in Table 1, we show a simple *Account Balance* dialogue with the required API interaction.

```
{"name":"Transfer",
"keys":"make.*transfer,send.*money...",
"confirmation":true,
"confirmation_question":"Do you confirm
    sending {Amount} to {ToAccount}?",
"execution_call":"execute_transfer",
"slots": [
   {"slot_name":"ToAccount",
        "slot_type":"StringList",
        "constraint":"_keys_:
            recipient_list",
        "question":"What is the name of
            the recipient?",
        "validation_api":"
            check_recipient",
        "error_message":{"val_err":"
            There is no saved recipient
            with the name you provided.
            Please enter one of the
            options among: "},
        "mandatory":true,
        "regex": "money to {slot}"},
   {"slot_name":"Amount",
        "slot_type":"Currency",
        "constraint":"€",
        "dependency":"FromAccount",
        "question":"How much money would
            you like to transfer?",
        "validation_api":"
            check_funds_limit",
        "error_message":{"val_err":"The
            amount cannot be processed,
            please specify a valid
            amount.", "fund_err":"Sorry,
            you do not have enough
            funds in your account. Would
            you like to change the
            amount?","limit_err":"Sorry,
            the amount you specified is
            above your transfer limit.
            Would you like to change the
            amount?"},
        "mandatory": true},
   {"slot_name":"NoteConfirm",
        "slot_type":"Confirmation",
        "constraint":"yes,ok,sure,
            correct,right,positive;no,
            not now,not today,negative;
            action:skipNext",
        "question":"Do you want to add a
            note to your transfer?",
        "validation_api":"NoValidate",
        "error_message":{"val_err":"I
            couldn't understand your
            request. Could you please
            confirm or reject adding a
            note?"},
        "mandatory":true},
   {"slot_name":"Note",
        "slot_type":"String",
        "constraint":"",
        "question":"Please type your
            note.",
        "validation_api":"savePost",
        "error_message":{"val_err":"
            There is a problem with your
            request. Your note could
            not be saved."},
        "mandatory":false }],
"success_message":"OK, your transfer is
    done. Can I help you with anything
    else?",
"error_message":"Your transfer cannot be
    executed. {ErrorMessage} Can I help
    you with anything else?"}
```

Listing 1: json FSDS for *Money Transfer*

4 System Evaluation

Since one of our main claims is that our abstraction framework can drastically reduce dialogue design and code complexity - as compared to OpenDial - we set up a comparison task. In this task we re-implemented three of the intent of the bank-

B:	Hello, how can I help you with ?
U:	I would like to check my account balance.
$\mathbf{A}_C$:	MESSAGE: ACCOUNT_LIST, INTENT: ACCOUNTBALANCE
$\mathbf{A}_R$:	[CHECKING, SAVINGS]
B:	You have Checking, Savings accounts. Which one would you like to query?
U:	savings, please
$\mathbf{A}_C$:	INFO_TYPE: CHECK_ACCOUNT, MESSAGE: SAVINGS
$\mathbf{A}_R$:	STATE: INFO_CHECK_SUCCESS
$\mathbf{A}_C$:	MESSAGE: EXECUTE_INTENT
$\mathbf{A}_R$:	MESSAGE: {AMOUNT:100, CURRENCYTYPE:€}, STATE: EXECUTE_SUCCESS
B:	The current balance on your Savings account is 100 €. Can I help you with anything else?

Table 1: An excerpt of an *Account Balance* dialogue generated from its FSDS toghether with the API calls. **B** is Bot, **U** is User, $\mathbf{A}_C$ is API call, $\mathbf{A}_R$ is API response.

ing scenario into the native OpenDial representation. The code reduction was of two orders of magnitude (on average from 2000 lines of XML code to 80 lines of json format FSDS description). Note that for this comparison we relied on a developer, so we excluded the time needed to learn the tool, that for OpenDial is expected to be much higher. As a second task, we ported the language-specific generic NLU/NLG definitions and original 9 Banking Domain intents to two new languages (Italian and Hungarian): on average setting up a completely functional dialogue agent with all 9 intents required from 3 to 4 hours. In this case we did not use a programmer but a native speaker per language with good knowledge of English.

5 Conclusion and Future Work

We presented FASTDial, a dialogue policy abstraction framework built on top of OpenDial that allows for fast prototyping and significant code complexity reduction in building conversational agents. We are planning to expand our framework with new features (e.g. new data types, multiple slots per turn), and also to build a version that allows for user driven dialogues. This new version would require some changes in the underlying logic. Moreover we are planning to integrate some available models for NLU and NLG into our FSDS structure to replace simple regular expression matching and template filling.

Acknowledgement

This work has been partially supported by the Conversational Banking Front-end EIT-Digital project (ID. 18039).

References

Antoine Bordes, Y-Lan Boureau, and Jason Weston. 2016. Learning end-to-end goal-oriented dialog. *arXiv preprint arXiv:1605.07683*.

Mikhail Burtsev, Alexander Seliverstov, Rafael Airapetyan, Mikhail Arkhipov, Dilyara Baymurzina, Nickolay Bushkov, Olga Gureenkova, Taras Khakhulin, Yuri Kuratov, Denis Kuznetsov, Alexey Litinsky, Varvara Logacheva, Alexey Lymar, Valentin Malykh, Maxim Petrov, Vadim Polulyakh, Leonid Pugachev, Alexey Sorokin, Maria Vikhreva, and Marat Zaynutdinov. 2018. Deeppavlov: Open-source library for dialogue systems. In *Proceedings of ACL 2018, System Demonstrations*, pages 122–127. Association for Computational Linguistics.

Matthew Henderson, Blaise Thomson, and Jason Williams. 2013. Dialog state tracking challenge 2&3.

Daniel Jurafsky and James H Martin. 2017. Dialog systems and chatbots. *Speech and language processing*.

P. Lison and C. Kennington. 2016. Opendial: A toolkit for developing spoken dialogue systems with probabilistic rules. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Demonstrations)*, pages 67–72, Berlin, Germany. Association for Computational Linguistics.

Samuel Louvan and Bernardo Magnini. 2018. Exploring named entity recognition as an auxiliary task for slot filling in conversational language understanding. In *Proceedings of the 2018 EMNLP Workshop SCAI: The 2nd International Workshop on Search-Oriented Conversational AI*, pages 74–80.

Grégoire Mesnil, Yann Dauphin, Kaisheng Yao, Yoshua Bengio, Li Deng, Dilek Hakkani-Tur, Xiaodong He, Larry Heck, Gokhan Tur, Dong Yu, et al. 2015. Using recurrent neural networks for slot filling in spoken language understanding. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 23(3):530–539.

Stefan Ultes, Lina M Rojas Barahona, Pei-Hao Su, David Vandyke, Dongho Kim, Inigo Casanueva, Paweł Budzianowski, Nikola Mrkšić, Tsung-Hsien Wen, Milica Gasic, et al. 2017. Pydial: A multi-domain statistical dialogue system toolkit. *Proceedings of ACL 2017, System Demonstrations*, pages 73–78.