DeRool: A rule-based dialog engine

Kevin Angele[1] and Jürgen Angele[2]

[1] Onlim GmbH
kevin.angele@onlim.com
http://onlim.com
[2] adesso, Competence Center Artificial Intelligence
juergen.angele@adesso.de
http://adesso.de

Abstract. Intelligent digital assistants as chatbots and voice assistants are already widely accepted and used. For the majority of the existing chatbots, the way people can communicate with them is still limited to some extent. In the future, the interaction between users and intelligent digital assistants should become more natural. The goal is to realize complex dialogs instead of independent question and answering parts. In real life, a dialog between humans is an ongoing change of the dialog initiative between the persons talking to each other. This natural way of communicating with each other needs to be implemented for intelligent digital assistants. The digital assistant, as the humans in a dialog, need to keep the information of the current conversation for the ongoing dialog and ask for information necessary to understand the user and fulfill their requests. To enable a more natural communication between users and assistants, we present DeRool, a rule-based dialog engine that is able to handle complex dialogs.

Keywords: Rule · OO-Logic · Dialog engine · Intelligent digital assistants.

1 Introduction

Intelligent digital assistants as chatbots and voice assistants are more popular than ever. Most people already used one of those to receive some sort of information or fulfill a specific task. Most of the existing chatbot and voice assistant solutions (e.g., Google Dialogflow[3]) are able to handle simple question-answer dialogs. The user asks a question, and the digital assistant returns an answer. The idea is that a conversation with a virtual digital assistant becomes more natural, like talking to a real person. Conversations between two persons are more than independent questioning and answering. Instead, it is more like an exchange of information and specific questions between those. Imagine a conversation in

[3] https://dialogflow.com/

a shop that sells particular sorts of wine. A conversation would start with the person who wants to buy a bottle of wine, giving necessary information to the salesperson. During the conversation, the salesperson will ask some questions to get an idea of the preferences of the person. In the end, the salesperson has all the information he needs to recommend a specific sort of wine that, in his eyes, fits best. Such conversations can not be handled by independent and straightforward questions and answers as the user would always need to mention all the facts he already provided again. The digital assistant needs, like the salesperson in the shop, to remember the information it already got from the user and ask questions to get all the necessary information that is still missing. In a nutshell, dialogs consist of an ongoing change of the dialog initiative between the user and the digital assistant retaining the context information.

To enable more natural conversations, we present a dialog engine based on rules called DeRool that allow us to define knowledge-based guided dialogs. Those dialog definitions are not done in a programmatical way, like using a programming language and compiling code to make the dialogs executable. They are defined in a declarative way using JSON syntax. An integrated interpreter then executes those definitions. One of the main benefits of the dialog engine is its independency from NLU frameworks. For example, IBM Watson[4] provides a function to model and execute dialogs. Still, it can only be used when using IBM Watsons NLU (natural language understanding) component to process the messages. DeRool acts on top of a triple store and has, therefore, access to the underlying data. All data that is necessary for the topics the digital agent should cover can be stored in the triple store and can also be used to guide the dialog in certain directions. Complex dialogs can be modeled using declarative rules. With those rules, it is also possible to include the underlying data into decisions made by the dialog engine. The dialogs can be extended by so called built-ins to access external service, to allow a dialog flow based on external information.

In the previous paragraph, we already mentioned that all the dialogs are defined using JSON syntax. JSON syntax is very popular for the development of APIs and in the web development domain. The dialog engine provides a basic structure on how a dialog needs to be defined and which properties need to be given to realize a dialog. To define pre- and postconditions OO-Logic [2] conditions are used. Due to those conditions, the dialogs can behave very flexible. Besides, also hardwired dialogs are possible. The transition between states is caused by incoming intents, that come from the NLU as a result of the processed user message. Every intent supported by the dialog engine needs a corresponding dialog state and action definition. The dialog engine is stateful, which means that all the intents and the current state for each user are stored while the dialog is active.

The paper is structured in the following way: In Section 2, we present the structure of dialog definitions, that are defined in JSON and an explanation of the available properties. Afterward, the processing of a certain intent is described. An example from industry is later used (Section 4) to show the capabilities and

[4] https://cloud.ibm.com/docs/assistant?topic=assistant-dialog-build

give an insight into the dialog definitions used by the dialog engine. Section 5 presents other approaches that support dialogs, and in the last section (Section 6), we conclude our paper and give an outlook on future work.

2 Dialog definition

A dialog definition follows a strict pattern that must be given in a document using JSON syntax. A dialog definition consists of four different parts:

- general information
- state definitions
- output of the current state definition
- response given by the user corresponding to the current dialog state

We start with the description of properties on the root level (*general information*), continue with the properties that are needed to define a specific dialog state (*state definitions*). Afterward, we present the properties that can be used to describe the appearance of the answer for the particular dialog state (*output of the current state definition*). Finally, properties that can be used for handling incoming intents as response to the output of the current dialog state are presented (*response given by the user*). Listing 1.1 shows an empty dialog definition pattern with all the possible properties that can be used for a dialog definition.

```
{
 "@id": "",
 "@type": "Dialog",
 "chatbots": [],
 "facts": [],
 "states": [{
  "state": "",
  "recognizes": [],
  "pre": "",
  "message": {
   "template": "",
   "type": ""
  },
  "queries": [],
  "response": [{
   "post": "",
   "recognizes": [],
   "goto": ""
  }]
 }]
}
```

Listing 1.1: Structure of a dialog definition

2.1 General information

Properties on the root level contain general information about the dialog. A unique identifier, the list of chatbots the dialog can be used for, and general

facts that need to be available in the dialog. Listing 1.2 shows the properties on the root level. In the following, we give a short description of each of those properties.

```
{
 "@id": "",
 "@type": "Dialog",
 "chatbots": [],
 "facts": "",
 "states": [],
}
```

Listing 1.2: Root level properties

- *@id* - is the unique identifier for the given dialog.
- *@type* - defines the type of the given JSON document. For a dialog definition this value is fix and always represented by "Dialog".
- *chatbots* - every intent that is sent to the dialog engine must provide a property called `chatbotId`. This property defines for which chatbots (`chatbotId`) the dialog is used.
- *facts* - it is possible to define facts that can be used in the dialog states. Those facts must be given in OO-Logic, e.g. *text(de, "Montag"). text(en, "Monday")*. Those facts are often used to support different languages.

Properties on the root level are extended by properties to define the dialog states.

2.2 Dialog state

The core of each dialog is the different dialog states a dialog can take. Each user message results in a state transition, even if the same state is entered again. Listing 1.3 shows the list of possible properties to define those dialog states.

```
{
 ...
 "states": [{
  "state": "",
  "recognizes": [],
  "pre": "",
  "message": {},
  "queries": [],
  "response": []
 }],
 ...
}
```

Listing 1.3: Dialog state properties

– *state* - a unique identifier for the corresponding state. The state identifier must be unique in the context of the given dialog definition. In different dialogs, it is possible to have states with the same name, but for a specific definition, it must be unique.
– *recognizes* - intents that should be recognized by the current state are defined in this property. It takes a list as different intents can trigger the same dialog state. The incoming intent must have an object as value for the property `parameters` and this object must have a property `intent_name` or `intent_id`.
– *pre* - OO-Logic condition that is checked before entering the state. If the condition is fulfilled, the given state will be entered.
– *message* - defines how the output of the current state looks like. It will be explained in the next subsection (Subsection 2.3).
– *queries* - it is possible to define queries that contain variables that are used for the current dialog state. This is used to fill the variables in the *response* part with the values of the underlying ontology. The queries are formulated in regard to the underlying ontology. Data stored in the triple store can be accessed that way.
– *response* - the *message* is returned to the user and the user gives a new input that is sent to the dialog engine. In the response object, it is defined how to handle the incoming user request as a response to the output of the current state. A further description can be found in Subsection 2.4.

In the following two subsections, we describe the *message* object to format the output that should be returned to the user and the *response* object to handle user requests that follow as an answer to those outputs.

2.3 Output of the current state definition

The output of a dialog state transition can either be a simple text or a more complex structured JSON/HTML object. The simple text can contain variables that must correspond to the variables used in the *queries* property (see the previous subsection). Those variables are then replaced with the result of the query, and the text is returned to the user. An example would be "My name is $NAME$", where the variable $NAME$ is then replaced by the value given in the result of the query. Listing 1.4 describes the JSON object structure to use more complex templates.

```
{
...
  "message": {
   "template": "",
   "type": ""
  },
...
}
```

Listing 1.4: Output definition properties

– *template* - name of a template must be given. This name corresponds to the file name of the template that should be used. Such a template contains variables as well that must correspond to the variables used in the query. After executing the query, the template is filled with the results of the query. It is also possible to receive multiple results from one query. In that case, the template is filled multiple times and combined into an array. This is, for instance, used to show a carousel of different information cards in the chatbot.
– *type* - the dialog engine currently supports templates defined in `HTML` or `JSON`.

2.4 Response given by the user

The dialog engine returns a response after a dialog state transition. The user might then send a new request to the dialog engine. A response object defines how to handle a response from the user for the current dialog state.

```
{
...
  "response": [{
   "post": "",
   "recognizes": [],
   "goto": ""
  }],
...
}
```

Listing 1.5: Response definition properties

– *post* - OO-Logic condition that is checked after the state transition. An example is given in the use case section (Section 4).
– *recognizes* - has the same meaning as the property on the state definition level. It defines the intents that are accepted as response to the output of the current state.
– *goto* - if the incoming intent matches one of the intents given in the *recognizes* property and the *post*-condition is fulfilled, this property defines the follow up state that should be entered.

Based on those patterns, it is possible to describe all kinds of dialog flows. As the queries and rules refer to an ontology, these dialog flows are knowledge-based. In the next section, we give a short introduction to the execution of dialogs in the dialog engine.

3 Dialog engine

When starting the triple store with the dialog engine extension, the dialog definitions and templates are loaded. Afterward, the dialog engine is ready and

waits for incoming intents. For an incoming intent, the dialog engine checks if there is already an active dialog for the current user. If that is the case, the intent is used to enter the next state (state transition). In case there is no active dialog for the user, the dialog engine checks if the given intent corresponds to a state, and the state is a start state. If a start state matches the given intent and the precondition of the state is fulfilled, this state is entered. For a state transition, if there is a pre-condition, it is checked if the condition is fulfilled. If yes, the state is entered, and otherwise, nothing happens, and the dialog is left. After a successful state transition, the query for the given state is executed. The result of that query is used to fill the response message/template. This message is sent back to the user, and the user can send a new message (is then resolved to an intent) as a response to the given output. For the new intent, it is checked if the response part of the current state expects the given intent. If the intent is recognized, the post-condition is checked, and the state identified by *goto* is entered. The dialog is left, if the post-condition is not fulfilled. Those steps are repeated until either the dialog is finished or the user leaves the dialog. The dialog engine handles the state of an arbitrary number of users. DeRool is based on SemReasoner, the successor of Ontobroker [1]. In [4], it has been shown that Ontobroker is the top performer. SemReasoner is, in most cases, twice as fast as Ontobroker.

4 PAYONE an in-use example from industry

DeRool is already used in chatbot related projects by larger companies. In the following, we present an in-use example for a provider for payment solutions in Germany, Austria, and Switzerland. First of all, we give some information on the company and their use cases. Afterward, we show an extract from the dialog definitions used to realize the use cases, and in the end, we present screenshots on real user interactions with the chatbot. At the time of writing the paper, the chatbot of this payment solution provider was realized as an MVP (minimal viable product) in preparation for a productive use at a later stage.

4.1 General

Payone GmbH (PAYONE)[5] is a company based in Frankfurt am Main that provides cashless payment solutions in Germany, Austria, and Switzerland. With 1,200 employees and about 400,000 customers, PAYONE processes roughly 2.6 billion payment transactions a year. The total payment volume is around 118 billion euros. Every transaction is processed within 100 milliseconds with a maximum of 12,000 transactions per second. PAYONE offers not only national and international payment methods but also risk management, debtor, and receivables management.

PAYONE has a call center for its customers that handles all concerns of customers. Especially the quantity of questions concerning bills, card transactions,

[5] https://www.payone.com/

and payouts is very high. For instance, each position in the bills for the customers is composed of several parts like payment terminal fees, fees for service fees, network fees, and fees for tax. There is only a restricted number of questions the customers have on those topics. On the other hand, the manual effort to answer those questions is relatively high, as the call center agents have to retrieve the relevant information from the operative IT-systems. This use case is perfect for an automatic interactive chatbot using dialogs.

There are different roles for DeRool in this case. DeRool, with its rule-based reasoner behind, serves as a semantic integration engine. It hosts an ontology describing bills, card transactions, and payouts with all their properties needed for the knowledge-based answering of the customer's questions. Attached to the ontology are the external information sources like an SAP system. The information inside SAP is brought to a semantic level by reinterpreting those data in terms of the ontology [3]. Attaching external information sources and integrating them into the ontology is done by rules as well. The second role of DeRool is providing a knowledge base. This knowledge base consists of the ontology and rules describing more complex relationships. Dependent on the available financial data, we developed a small ontology that describes bills, positions in bills, and the different kinds of fees. More complex relationships are described using rules like the amount of a payout is given by the sum of the amounts of the single pay positions:

```
NettoSumme: ?A[NettoSumme: ?NS] :- ?NS := sum{?N [?A] | ?A:Auszahlung, ?A[
    Position: ?P], ?P[Nettowert: ?N]}.
```

Finally, the third role of DeRool is handling the different dialog states and creating knowledge based (rule and query-based) answers to the customers like it has been described above.

4.2 Extract of the dialog definitions

In the following, we show some extracts of the dialog definitions for our use case. There are three different main branches for the dialogs: *bills, card transactions*, and *payouts*. We will show two dialog steps from the branch card transactions. In the dialog, the user is first asked for an SMS code sent to his mobile phone as part of the two-factor authentication. Then the chatbot asks for the amount and the date of the transaction. After the user has provided this information, the chatbot assures this information and generates the response, which explains the different parts of that position. Finally, the user is asked whether the respective document should be sent to the user by e-mail, and finally, the user confirms that. In listing 1.6 the entry dialog for this branch is given. As soon as the NLU (natural language understanding) recogizes the intent *af06c63b-e726-4302-9082-4b5cedaacbe3* the dialog engine jumps into that state and asks the user for the amount and the date of the transaction. If that amount can be found (first *post* condition) the dialog engine jumps to state 1 which is shown in listing 1.7.

```
{
"state": "0",
```

```
  "recognizes": ["af06c63b-e726-4302-9082-4b5cedaacbe3"],
  "message": "Nennen Sie bitte den Kaufbetrag und das Kaufdatum für die gew
      ünschte Transaktion.",
  "response": [{
    "recognizes": ["edd4c4fa-00fe-481d-a11c-900f88e0de7f"],
    "post":"CardPayoment[amount:?M],?A:Auszahlungsposition, ?A[Nettowert: ?M]
        .",
    "goto": "1"
  },
  {
    "recognizes": ["edd4c4fa-00fe-481d-a11c-900f88e0de7f"],
    "post":"CardPayoment[amount:?M],?A:Auszahlungsposition, ?A[Nettowert: ?N]
        , ?M != ?N.",
    "goto": "10"
  }
]
}
```

Listing 1.6: First payment transaction dialog state

```
{
"state": "1",
  "queries":[
     "Amount: ?- CardPayment[amount:?M],
              ?A:Auszahlungsposition,
              ?A[Nettowert:?M]."
],
  "message": {
    "Wir haben mehrere Beträge in der Höhe von $M$  gefunden. Bitte geben Sie
        die letzten 4 Ziffern der Kartennummer ein"},
  "response": [{
    "recognizes": ["fb387d82-5acb-4f46-a510-91482ea3a847"],
    "post": "CardPayment[Cardnumber:?C,amount:?M],
            ?A:Auszahlungsposition,
            ?A[Nettowert:?M,Kartennummer:?N],
            ?C = ?N",
    "goto": "2"
   },
   {
    "recognizes": ["fb387d82-5acb-4f46-a510-91482ea3a847"],
    "post": "CardPayment[Cardnumber:?C, amount:?M],
            ?A:Auszahlungsposition,
            ?A[Nettowert:?M,Kartennummer:?N],
            ?C != ?N",
    "goto": "1"
   }
 ]
}
```

Listing 1.7: Second payment transaction dialog state

In listing 1.7, the second state of this dialog is given. It is the part of the dialog where the chatbot assures the input given by the user. In the field *message*, the bot says, "Wir haben mehrere Beträge in Höhe von M gefunden. Bitte geben Sie die letzten 4 Ziffern der Kartennummer ein". The variable M is filled by the OO-logic query *Amount*. The query refers to the property *Nettowert* of a class *Auszahlungsposition* in the ontology. If the card number given by the user is the same as the card number for the transaction (see the first *post* condition), the dialog state changes to state 2, otherwise if it is not the same, we go back to state 1.

4.3 Example chatbot conversation

As mentioned at the beginning of this section, the outcome of the PAYONE project is a chatbot that can handle questions concerning bills, card transactions, and payouts. The following screenshots give an impression on the card transaction dialog. In Figure 1 the user requests information for a card transaction ("Kartenzahlung"). To be able to help the user, the chatbot needs more information like the amount and the date the transaction was done ("1072,50, 01.11.18"). Based on this information DeRool searches for a transaction that matches. If there is more than one transaction, the user needs to enter the last digits of his card number.

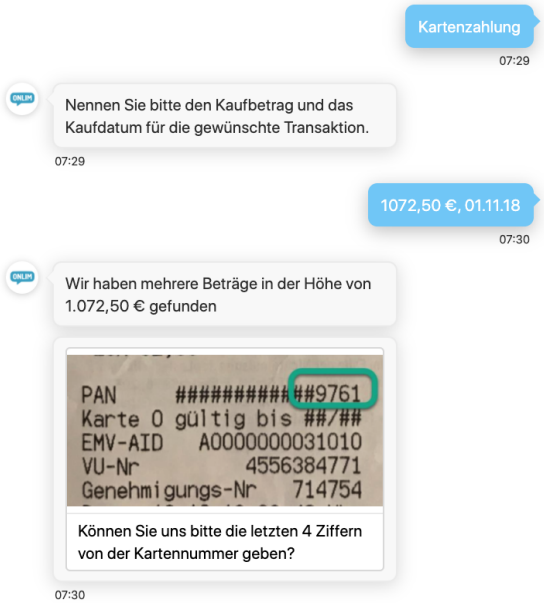

Fig. 1: Card transaction dialog

After the user entered the missing information ("4511", see Figure 2), DeRool searches again for matching transactions. If there is a transaction matching all given information ("1072,50, 01.11.18, 4511"), details on this transaction are presented to the user. The chatbot provides some basic information for the user and in addition a web link to a more detailed description.

The space for displaying information in a chatbot is restricted not to overwhelm the user. Therefore, detailed information on transactions are available on a separate website.

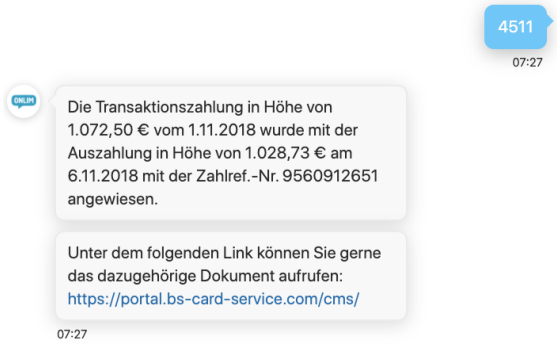

Fig. 2: Transaction details

5 Related Work

Many small companies offer chatbot solutions and platforms for intelligent digital assistants. There are only a few big players like Amazon with Alexa[6], Google with Dialogflow[7] and also IBM with Watson[8]. In addition to those, we will present in this section mercury.ai[9] a platform that provides a dialog management component.

Amazon Alexa is a voice assistant developed by Amazon. The platform can be used by developers to build a custom skill for free. This skill can then be used on all the Amazon devices. So far, Alexa supports only simple question answering and no guided dialogs at all.

Google uses his platform Dialogflow for the digital assistant called Google Assistant. Google itself develops Dialogflow. Regarding dialogs, Dialogflow allows storing certain context information that can be used either as input or as output for an intent. Also, it supports follow up intents that can be triggered based on context information. For very simple dialogs like collecting a set of information, Dialogflow allows to define prompts. Those will be sent to the user when information is missing. Dialogflow has a very developer-friendly user interface that makes it easy to define and configure intents and simple dialogs. On the other hand, more complex dialogs are not supported and can not be modeled with Dialogflow. Dialogflow can not use data for knowledge-based control of the dialog flow and neither for generating knowledge-based answers.

With IBM Watson, it is possible to define more complex dialogs based on user input. Each dialog state has a specific entry condition that is either based on the recognized intent, entity, or a particular context. Also, Watson provides the possibility to define follow up states that use the context set from the previous state. Those follow up states are again triggered based on the conditions

[6] https://alexa.amazon.com
[7] https://dialogflow.com/
[8] https://ibm.com/watson
[9] https://mercury.ai

specified. As well as Dialogflow Watson provides so-called slots to prompt for required information and stores them in the context to be used by other follow up states. Watson's user interface shows all the connections between different dialog states in a tree view. This makes it very easy for dialog designers to model dialogs. On the other hand, those dialogs can again only be defined based on user input. There is no possibility of including external services or the underlying data for controlling the dialog flow or for providing knowledge-based answers.

A quite young company providing a chatbot platform is mercury.ai, founded in 2016. So far, they don't support more than Dialogflow or Watson. Nevertheless, in their documentation, they mention that the Dialog Engine they are building will, at some point, be able to handle data-driven dialogs. So far, to the best of our knowledge, we didn't find any data-driven dialog example for the mercury.ai platform.

6 Conclusion

We presented DeRool, a dialog engine based on OO-Logic rules. The goal of DeRool is to enable more natural conversations between users and intelligent digital assistants (like chatbots or voice assistants). To understand the flexibility DeRool provides, we presented the format for the definitions of dialogs and explained how the dialog engine executes the dialogs. DeRool is used in a project at PAYONE, a provider for payment solutions acting in the DACH (German, Austria, and Switzerland) region. They use the dialog engine to help users with questions on bills, card transactions, and payouts. One part of the use case was an extract of those dialog definitions and an example conversation a user could follow in the chatbot. DeRool is used productively at Bitmarck, a large IT service provider for health insurance in Germany. In the future, we want to develop a tool that supports the process of modeling dialogs with a user-friendly interface. This tool should also verify the dialog definitions and present errors before they occur during the execution of the dialogs. In the end, the tool is also able to connect to NLP tools (like Dialogflow[10] and many more) to easily link the intents defined in those platforms with the dialog states they correspond to.

References

1. Angele, J.: Ontobroker. Semantic Web **5**(3), 221–235 (2014)
2. Angele, J., Angele, K.: Oo-logic: a successor of f-logic. International Joint Conference on Rules and Reasoning (2019)
3. Angele, J., Gesmann, M.: Data integration using semantic technology: a use case. In: 2006 Second International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML'06). pp. 58–66. IEEE (2006)
4. Liang, S., Fodor, P., Wan, H., Kifer, M.: Openrulebench: an analysis of the performance of rule engines. In: Proceedings of the 18th international conference on World wide web. pp. 601–610 (2009)

[10] https://dialogflow.com/