An adaptable task-oriented dialog system for stand-alone embedded devices

Long Duong, Vu Cong Duy Hoang, Tuyen Quang Pham, Yu-Heng Hong, Vladislavs Dovgalecs, Guy Bashkansky, Jason Black, Andrew Bleeker, Serge Le Huitouze, Mark Johnson
Oracle Digital Assistant
first.last@oracle.com

Abstract

This paper describes a spoken-language end-to-end task-oriented dialogue system for small embedded devices such as home appliances. While the current system implements a smart alarm clock with advanced calendar scheduling functionality, the system is designed to make it easy to port to other application domains (e.g., the dialogue component factors out domain-specific execution from domain-general actions such as requesting and updating slot values). The system does not require internet connectivity because all components, including speech recognition, natural language understanding, dialogue management, execution and text-to-speech, run locally on the embedded device (our demo uses a Raspberry Pi). This simplifies deployment, minimizes server costs and most importantly, eliminates user privacy risks. The demo video in alarm domain is here youtu.be/N3IBMGocvHU.

1 Introduction

Communicating directly using voice is a more natural way to interact with computer and household appliances. People already interact with smart appliances such as microwaves and alarm clocks using voice control. However, these devices need to connect to cloud services to process user requests. We focus on building entire task-oriented dialog applications on cheap edge devices such as the Raspberry Pi [1] which operate independently of any internet connection. This approach: **a)** ensures user privacy, **b)** abolishes server costs, and **c)** eliminates network connection latency. Specialized neural network chips and generic embedded CPU devices are becoming significantly cheaper, making voice interfaces price-competitive with display-based controllers. Our vision is that in the next few years, AI-powered devices will be in appliances throughout everyone's home. This paper describes an end-to-end smart alarm clock demo run offline on a small device as a proof of concept for our vision. The approach proposed in this paper is general and easily adapted to different languages and domains.

We describe how we meet the challenges of implementing a complete speech-based task-oriented dialogue system on a small embedded device with low memory and computational power. Our design makes no assumption on the availability of peripherals such as a display screen or buttons for user responses. Just as in many cloud-based dialogue systems, our system is a pipeline of standard components, including a wake word detector, automatic speech recognition (ASR), natural language understanding (NLU), dialogue manager (includes dialogue state tracker and dialogue policy, and execution), natural language generator (NLG) and text-to-speech (TTS). Figure 1 shows the overall organisation of these components. ASR converts spoken user requests to text, which is then fed to NLU components consisting of a Named Entity Recogniser (NER) and a Semantic Parser. The NLU output is a *logical form* (LF1) which encodes the current user request. The Dialogue State Tracker (DST) integrates LF1 with the previous dialogue states and dialog acts to produce an updated dialogue state (LF2). While LF1 only represents a single dialogue turn, LF2 represents the entire dialogue prior to this point in time. The domain-specific Execution component executes LF2, and the results of Execution are returned to Dialog Policy component and also saved to Context Stack which contains all intermediate results. The Dialogue Policy uses the execution results and LF2 to produce a dialogue act response (LF3), which is also recorded in the Context Stack. LF3 is used to generate output to the user which is converted to speech using a Text-To-

[1] https://www.raspberrypi.org/

Speech (TTS).

Given the hardware constraints of embedded devices, we decided to use rule-based approaches where possible, and to reserve classifier-based and deep learning approaches for components such as the NER and the Semantic Parser, where linguistic variation and construction would be difficult to capture with hand-written rules. We use a rule-based Dialogue Manager and a template-based NLG for this reason. To make it easier to adapt the system to new domains and languages, the domain-specific code is concentrated in the ASR, NER, Semantic Parser, Execution and NLG components. The system is implemented in C++11 to simplify deployment on embedded devices.

2 Logical Forms Design

Information is exchanged between components using representations that we call Logical Forms (LFs). A variety of logical forms have been proposed in the literature, such as lambda.DCS and the lambda calculus (Zettlemoyer and Collins, 2005). Intent plus slots representations are standard in many dialog systems, but they cannot express complicated scenarios involving conditionals, nested structures, multi-intents and quantifier scope.

Our LFs are JSON objects[2] that we call Topic-Action Attribute-Value Logical Forms (TAVLFs). These are attribute-value structures (Johnson, 1988) whose organisation is inspired motivated by CUED standard dialog acts (Young, 2007).

At the top level, TAVLFs have a bipartite structure consisting of topic and action attributes. The topic identifies the primary entities under discussion, while the action specifies what the user requests the system to do with these entities. For example, the request *Move my work out alarm tomorrow 1 hour earlier* is translated to TAVLF:

```
{"topic": { "name": "work out"},
  "action": { "edit": `
      {"offset_direction": "
         earlier",
     "offset_time": "1 hour"}}}
```

Here the topic attribute selects calendar entries that satisfy "name":"workout". The action attribute specifies what the system should do to the Topic entities; in this case apply the edit action with arguments offset_time and offset_direction.

[2]www.json.org

TAVLFs can express complicated use cases such as multi-intent requests, nested finds, conditional requests, as well as quantifiers and superlatives. The bipartite separation into Topic and Action makes it easier to handle follow-up requests, since it is likely that the next utterance will involve the previous topic. We explain how we handle follow-up utterances in more detail in Section 5. Thus we demonstrate that embedded systems, despite limitations in both memory and computational power, can handle complicated utterances.

3 Wake Word Detection and Automatic Speech Recognition

We use similar technology for both Wake Word detection and ASR. After a wake word is detected, the ASR is activated to convert the following speech into text. The user needs to wake the system for each utterance, except for cases where the system requests a response from user; e.g., no wake work is required when system asks for additional information. Porting an ASR system to a small embedded device is challenging. After evaluating a variety of approaches we decided to use a DNN-based acoustic model together with a fast HMM-based language model decoder. This permits us to easily customise the ASR vocabulary for a new domain. We developed a customised version of Kaldi (Povey et al., 2011) which achieves real time factor of 0.23 even on a small embedded device. The ASR also provides a confidence score based on the HMM posterior probability, which the Dialog Manager uses to detect likely cases of ASR failure.

4 Natural Language Understanding

The Natural Language Understanding (NLU) component translates each utterance into its corresponding logical form (LF1). There are two steps to this process: Named Entity Recognition (NER) and seq2seq based Semantic Parsing. First, the user utterance is NER-tagged and delexicalised (i.e., named entities are replaced with their named entity types). The delexicalized utterance forms the input to the seq2seq semantic parser, which produces delexicalized logical form. This logical form is relexicalized (i.e., the named entity types are replaced with the original named entities) to produce the output logical form (LF1). Figure 2 shows the NLU pipeline. We use a CRF tagger based on CRFSuite (Okazaki, 2007) for NER, and

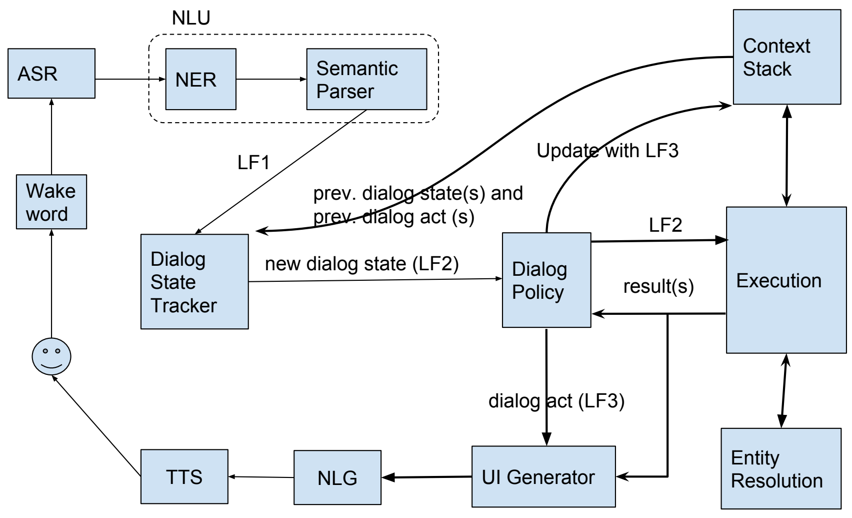

Figure 1: End-to-end embedded dialogue system architecture.

User	make my work out alarm 1 hour earlier
NER	make my <NAME>work out</NAME>alarm <TIME_SPAN>1 hour</TIME_SPAN>earlier
Delex	make my <NAME>#0 alarm <TIME_SPAN>#0 earlier
Seq2seq	{"action":{"edit":{"offset_direction":"earlier","offset_time":"<TIME_SPAN>#0"}}, "topic":{"name":"<NAME>#0"}}
Relex (LF1)	{"action":{"edit":{"offset_direction":"earlier","offset_time":"1hour"}}, "topic":{"name":"work out"}}

Figure 2: The NLU pipeline that maps utterances to Logical Forms (LF1).

employ a deep learning seq2seq model for Semantic Parsing.

4.1 Semantic Parser

Our Semantic Parser is based on the dual-RNN sequence-to-sequence architecture with attention originally proposed for neural machine translation (Bahdanau et al., 2014). We use it to generate Logical Forms as in Dong and Lapata (2016). The seq2seq model also generates a log loss confidence score, which the Dialog Policy Manager uses to detect likely Semantic Parser errors (see section 5).

It is challenging to fit a seq2seq model into a small embedded device. We solve this problem by extensive hyper-parameter tuning using the successive halving process proposed in Hyperband (Li et al., 2016). This approach enables us to explore a large number of hyper-parameter configurations quickly. We randomly generate around a thousand different configurations, which vary the source and target cell architectures (e.g. LSTM, GRU), number of layers, learning rates, drop out rate and mini batch size. We measure memory usage and latency as well as accuracy on the development set. We select the hyper-parameter configuration with the highest dev set accuracy that satisfies our memory constraints and has the acceptable latency (usually 100 ms/utterance). Models are trained using Tensorflow on a GPU cluster. Trained models are quantized and exported to our C++ runtime.

One of the challenges in building a semantic parser is obtaining suitable training data. We adapt and extend the crowd-based "overnight" approach of Wang et al. (2015) by adding an additional *validation task*, where other crowd workers validate the paraphrases from the *paraphrase task*. We run the validation task in real-time so we can provide on-line bonuses or penalties, which dramatically reduces spam and improves paraphrasing quality.

5 Dialogue Management

Dialogue management is central to any task-oriented dialog system. It is responsible for Dialogue State Tracking, executing the task (Execution), and determining how to interact with the user (Dialogue Policy).

5.1 Dialogue State Tracking

The Dialogue State Tracking (DST) component combines LF1 with information from the dialogue

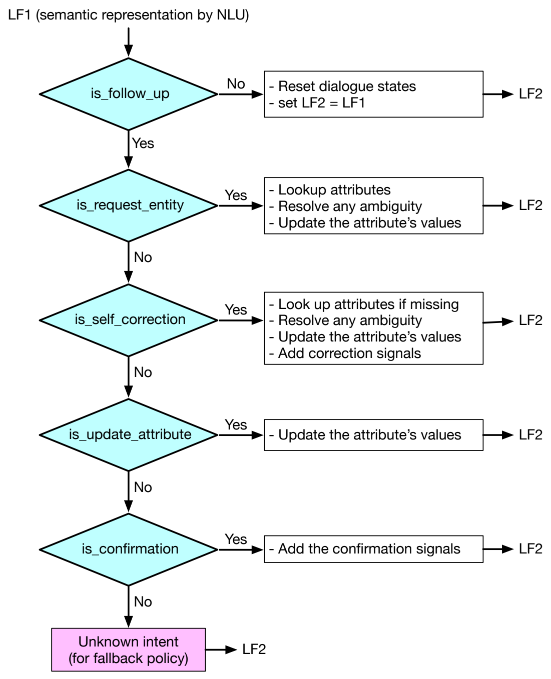

Figure 3: Logics for rule-based DST.

context (previous dialogue states and acts) to compute a Logical Form representation LF2 of the entire dialogue so far. We adopted a rule-based approach for the DST component because it: **a)** is easy to implement, **b)** requires no data, **c)** is extremely fast at run time, and **d)** provides an easy way to incorporate domain-specific information.

The high-level organisation of our rule-based DST is shown in Figure 3 (Appendix A.1). First, the DST distinguishes between follow-up and root (or non-follow-up) utterances by inspecting the Semantic Parser output LF1. If the utterance is a root utterance, the DST sets LF2 to be LF1 and resets the current dialogue state context to start a new conversation. The DST distinguishes four different kinds of follow-up utterances by inspecting LF1: *is_request_entity*, *is_self_correction*, *is_update_attributes*, and *is_confirmation*. Section A.1 presents an example of how the DST functions in alarm clock domain.

5.2 Execution and Dialogue Policy

In our system, execution and dialogue policy work closely together. The Dialogue Policy component takes LF2 as input and passes it to the Execution component. The Execution component is responsible for actually executing user requests; in our system it translates them into SQL queries and executes against a database, producing a set of execution results. The Execution component interacts with Named Entity Resolution if any named entity string in LF2 is not exactly matched in the database for retrieval. For example, the named entity ”7 pm” must match the time 19:00 in the database.

Our system consists of a largely application-independent Dialogue Policy component and an application-specific Execution component. Execution is typically domain specific because it requires specific knowledge about the application. The Dialogue Policy component uses the execution results to generate the system response, which is encoded as a LF3. The Dialogue Policy component is associated with a set of types that encode the different kinds of information that the Execution component can return. For instance, if the execution is successful, the Dialog Policy needs to inform the user of the Execution results. But if there are execution errors (e.g., because the request is lacking essential information) the Dialogue Policy component may request additional information or clarification. Our Dialog Policy component uses 8 domain-independent execution return types (see our table 1 in Appendix A for more detail). For each type, the dialogue act (LF3) is constructed accordingly. By separating the Dialogue Policy and Execution components we make it easier to port our system to new applications.

We use the NLU and ASR confidence scores to trigger fall-back dialog policies that vary based on the kind of error we believe has occurred. For example, the system might ask user to speak more clearly if ASR confidence score is low, or to express the request differently if NLU score is low. We set the thresholds for each component using development data. The Dialog Policy component is also update the Context Stack, which stores all the information from LF2, the execution results, and the dialog act for current dialog turn.

6 Other components

6.1 UI Generator

Our modular design includes a User Interface (UI) component, which is responsible for the user interface. The UI depends on the device hardware, e.g., touch screen, buttons etc. Because our current system uses speech input and output, the UI component directly passes the dialog act (LF3) to the NLG component.

Dialog Act	Template 1	Template 2 ...
{inform:{count:0,when_date:X}}	There aren't any alarm for {X}	no alarm for X
{inform:{count:C,when_date:X}}	There are {C} alarms for {X}	...
{request:{confirm:{}}}	Are you sure?	Can you confirm?

Figure 4: Example NLG templates for alarm clock domain. Our templates are delexicalized; C, X, Y are variables which will be replaced with real values. Multiple variants are provided for each schematic LF to increase the diversity of the generated output.

6.2 Natural Language Generator

We use a template-based NLG, which translates dialog acts (LF3) produced by the Dialog Policy component into text that the TTS system can pronounce. Figure 4 shows some example templates. We use a hash function for efficient template retrieval. If multiple templates are found, we prefer the best match. For example, the dialog act *inform:{count:0,when_date:tomorrow}* matches the first two templates in Figure 4, so the first one is selected because the value of *count* attribute matches exactly.

6.3 Text to Speech

We need a TTS engine that is lightweight and fast enough to run on embedded devices. Open source TTS systems based on deep learning technology, such as Tacotron2 (Shen et al., 2017) and DeepVoice3 (Ping et al., 2017), produce high quality output but very slow on embedded devices. Other open source TTS that use HMM-based synthetic voices, such as MaryTTS (Charfuelan and Steiner, 2013) or Mimic [3], are fast but either of low quality or are difficult to port to embedded devices. We decided to use a commercial embedded TTS solution targeted at embedded devices.

7 Case Study: Alarm Clock Showcase

We built an alarm clock application to showcase our system. The application supports features such as create, delete, cancel, edit and snooze alarm, with attributes such as date, time, day and name. It also provides more advanced features such as conditionals, negation and multi-intent requests. It handles a variety of dialog use cases, such as request for confirmation, request for additional information, provide suggestions and inform about invalid values.

The alarm bot is deployed on a Raspberry Pi 3+ with Cortex-A53 CPU at 1.4GHz clock rate and 1GB Ram, which currently costs \$35 not including microphone and speaker. The NER and Semantic Parser is trained on $\approx$11k paraphrases of more than 1k Logical Forms, which we collected using our extension of the “overnight” process. The exact match accuracy on development set which is randomly sampled from training set is 85%. Hyper-parameter tuning using HyperBand searched 400 configurations to find the highest accuracy model with a maximum latency of 100ms on the target device. The Semantic Parser model size is 2.5 MB, while the NER model size is 0.4 MB; these consume 15.6 MB and 0.4 MB RAM at run time respectively. The ASR acoustic model size is 7.9 MB and SLM takes 47MB on disk. End to end examples with intermediate results can be found in Appendix A.3

8 Conclusion and Future Work

We presented a full end-to-end task-oriented dialog system that can be deployed on a cheap embedded device. The proposed framework is sufficiently general for rapid adaptation to new domains and languages. We demonstrate the capabilities of our system with an alarm clock application that can understand complicated user requests and handle complex dialog use cases. In future work we plan to improve the robustness of the whole pipeline by using pretrained embeddings for semantic parser, and investigate combining the NER, Semantic Parser and DST into a single deep learning model.

[3]https://mimic.mycroft.ai/

References

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *CoRR*.

Marcela Charfuelan and Ingmar Steiner. 2013. Expressive speech synthesis in MARY TTS using audiobook data and EmotionML. In *Interspeech*, pages 1564–1568, Lyon, France.

Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *ACL*, pages 33–43.

Mark Johnson. 1988. *Attribute Value Logic and The Theory of Grammar*. Number 16 in CSLI Lecture Notes Series. Chicago University Press, Chicago.

Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2016. Efficient hyperparameter optimization and infinitely many armed bandits. *CoRR*.

Naoaki Okazaki. 2007. Crfsuite: a fast implementation of conditional random fields (crfs).

Wei Ping, Kainan Peng, Andrew Gibiansky, et al. 2017. Deep voice 3: 2000-speaker neural text-to-speech. *CoRR*.

Daniel Povey, Arnab Ghoshal, Gilles Boulianne, et al. 2011. The kaldi speech recognition toolkit. In *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*.

Jonathan Shen, Ruoming Pang, Ron J. Weiss, et al. 2017. Natural TTS synthesis by conditioning wavenet on mel spectrogram predictions. *CoRR*, abs/1712.05884.

Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a semantic parser overnight. In *ACL*, pages 1332–1342, Beijing, China.

S. Young. 2007. Cued standard dialogue acts. Technical report, Cambridge University Engineering Dept.

Luke S. Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*, pages 658–666.

A Appendices

A.1 Dialog state tracking examples

After recognizing an utterance as a follow-up, we further classify the utterance into the following categories:

Request Entity is for utterances which provide a value, but do not specify the attribute or slot the provided value fills. The dialog state tracker decides which attribute the value belong to based on an ontology learned from semantic parser training data. For example.

1. User: Wake me up tomorrow
2. System: what time?
3. User: 8 am

This utterance is classified as *request entity* because it does not specify which attribute or slot the entity “8 am” fills. The corresponding logical form (LF1) for that utterance is:

```
{"action":{"follow_up":{"entity":
   "8 am"}}}
```

Self Correction is for utterances where the user corrects values they have previously supplied. For example:

1. User: Wake me up tomorrow at 8 am.
2. System: Done, you alarm at 8am tomorrow has been set.
3. User: Sorry make it 9 am please

The last utterance will be recognised as a *self correction* by the Semantic Parser. We execute this by rolling back the database execution, modifying the required value (i.e. from 8 am to 9 am) and executing the new logical form LF2. The corresponding logical form (LF1) is:

```
{"action":{"follow_up":{"entity":
   "9 am","self_correction":true}
   }}
```

And LF2 is:

```
{"action":{"create":{"when_day":"
   tomorrow","when_time":"9 am","
   self_correction":"true"}}}
```

Update Attribute is for utterances where the semantic parser can identify which attribute the user is referring to. For example,

1. User: create an alarm for tomorrow at 10 am.
2. System: Done, your alarm has been created.
3. User: call that alarm “meeting with Julie”.

The semantic parser can extract the attribute (i.e. `name` in this example) and associated value (i.e. *meeting with Julie*) from the last utterance. Updating an attribute is a standard operation in dialog state tracking. The corresponding logical form (LF1) is:

```
{"action":{"follow_up":{"
   attribute":{"name":"meeting
   with Julie"}}}}
```

Confirmation is for utterances that semantic parser recognizes as a confirmation. The last utterance in the following dialog is an example of a confirmation:

1. User: Delete my alarm for tomorrow morning
2. System: You have 2 alarms for tomorrow, do you want to delete those?
3. User: Yes, do it.

The corresponding logical form (LF1) is

```
{"action":{"follow_up":{"
   confirmation":"yes"}}}
```

A.2 Execution return types

See table 1.

A.3 End-to-end examples

```
User: hey alarm clock, wake me up
     tomorrow
LF1 = LF2 :  {"action":{"create":
   {"when_day":"tomorrow"}}}
Execution: {"execution_results":[
   {"action":"create","error_code
   ":1,"error_attributes":["
   when_time"],"results":[]}]}
LF3: {"policy":[{"request":{"
   when_time":{}}}]}
NLG: When would you like it to
   ring?
-------------------
User: 6 am please
LF1: {"action":{"follow_up":{"
   entity":"6 am"}}}
LF2: {"action":{"create":{"
   when_day":"tomorrow","
   when_time":"6 am"}}}
Execution: {"execution_results":[
   {"action":"create","error_code
   ":0,"error_attributes":[],"
```

Return Types	Description
Execution success	Execution finishes successfully.
Expect zero got more	Execution expects no entity but got more. For example, create database entries that already exist.
Expect one got more	Execution expects exactly one entity but got more. For example, edit alarm by name but there are two alarms having the same name.
Expect at least one got zero	The value is not found. For example, query alarms by name but there aren't any alarms matching that name
Invalid values	The value is invalid. For example, create an alarm in the past.
Missing attributes	Missing attributes for execution. For example, create an alarm without specifying time.
Yes/no confirmation	Execution pauses to wait for confirmation from user for critical actions such as purchase, delete etc.
Execution fail	Execution fail for unknown reasons. This is useful for fallback policy.

Table 1: Execution return types.

```
       results":[{"id":"1","when_date
       ":"2019-04-11","when_time":"06
       :00"}]}]}
11  LF3: {"policy":[{"inform":{"
       action":"create","when_date":"
       2019-04-11","when_time":"06:00
       "}}]}
12  NLG: Your alarm will go off at 06
       :00 tomorrow.
```

LF1 and LF2 in line 2 above are the same because this is the start of a dialog, so the DST just passes through the LF1. The execution (line 3) returns error code = 1 (missing attribute). The user provides the missing information (line 7) and execution is successful (error code = 0 in line 10).

The following is a more complicated dialog with negation and self correction.

```
1   User: hey alarm clock, could you
       create alarms called feed the
       dog for every weekdays except
       Friday at 8 am
2   LF1 = LF2 :  {"action":{"create":
       {"interval":"1w","name":"feed
       the dog","when_day":["weekdays
       ","~friday"],"when_time":"8 am
       "}}}
3   Execution: {"execution_results":[
       {"action":"create","error_code
       ":0,"error_attributes":[],"
       results":[{"id":"2","interval"
       :"1w","name":"feed the dog","
       when_day":"monday,tuesday,
       wednesday,thursday","when_time
       ":"08:00"}]}]}
4   LF3: {"policy":[{"inform":{"
       action":"create","interval":"1
       w","name":"feed the dog","
       when_day":"monday,tuesday,
       wednesday,thursday","when_time
       ":"08:00"}}]}
5   NLG: Your weekly alarm called
       feed the dog will ring every
       monday, tuesday, wednesday and
        thursday at 08:00.
6   --------------------
7   User: hey alarm clock, sorry i
       meant 9 am
8   LF1: {"action":{"follow_up":{"
       entity":"9 am","
       self_correction":true}}}
9   LF2: {"action":{"create":{"
       interval":"1w","name":"feed
       the dog","when_day":["weekdays
       ","~friday"],"when_time":"9 am
       ","self_correction":"true"}}}
10  Execution: {"execution_results":[
       {"action":"edit","error_code":
       0,"error_attributes":[],"
       results":[{"id":"2","interval"
       :"1w","name":"feed the dog","
       when_day":"monday,tuesday,
       wednesday,thursday","when_time
       ":"08:00","when_time_new":"09:
       00"}]}]}
    LF3: {"policy":[{"inform":{"
       action":"edit","interval":"1w"
       ,"name":"feed the dog","
       when_day":"monday,tuesday,
       wednesday,thursday","when_time
       ":"08:00","when_time_new":"09:
       00"}}]}
    NLG: Your recurring monday,
       tuesday,wednesday and thursday
        alarm called feed the dog has
        been moved from 08:00 to 09:0
       0
```