VADiMoS: A Web Tool for Designing Voice Assistant Independent and Ontology Based Dialogs

Thomas Ranzenberger, Christian Hacker

Elektrobit Automotive GmbH, Erlangen, Germany
{thomas.ranzenberger, christian.hacker}@elektrobit.com

Abstract: The development of custom skills for the popular Voice Assistants (VAs) of Google and Amazon is usually done in their corresponding eco system for the specific VA. However, the expectations to a voice assistant are very high: natural language understanding, open domain, task oriented, and smart dialogs resulting in correct and context dependent responses. The eco system of the VAs is often not capable to handle such natural dialogs. It is necessary to code simple dialogs running in the backend of the voice assistant, to learn dialogs based on a huge set of domain data, or to design dialogs based on rules. In our approach we design a knowledge graph based on objects and their relations. This knowledge represents the domain, the dialogs, language resources, and rules. A rule engine operates on this dynamic knowledge base and calculates the next dialog step. This approach is voice assistant independent and can be combined with the top on-market VAs, like Alexa or Google Assistant. In this paper we present our web-based Voice Assistant Dialog Modeling Service (VADiMoS) which enables such ontology based dialog modeling, testing, simulation, and deployment to the VA. VADiMoS abstracts the complex definition of dialog rules which are part of our ontology by providing a template-based rule editor. This enables the interactive and test-driven creation of human machine dialogs.

1 Introduction

Voice assistants (VAs) become more and more present in our lives. TV, smart speakers, and the complete home automation can be easily controlled via voice. We expect that popular VAs like Alexa and Google will also replace more and more the generation of inflexible dialog systems in our cars, where only few dedicated commands lead to success. We want to access car related functions or travel related information with natural speech and don't want to to without all the voice interaction we are used to have at home.

The development of custom dialogs, actions, or skills for the popular VAs is usually done in their corresponding eco system for the specific VA. We described in [1] an ontology-based voice dialog framework for in-vehicle infotainment. The system was specified by a declarative model comprising the dialog definition, dialog management logic, and domain knowledge, which are backed by an RDFS ontology. The approach simplifies coping with dialog phenomena such as anaphora or implicit confirmation, that can be addressed without writing code.

In this paper we present our web-based Voice Assistant Dialog Modeling Service (VADiMoS) which enables VA independent modeling, testing, simulation of dialogs, and deployment to the top on-market VAs, like Alexa or Google Assistant. The dialog management approach uses an ontology (e.g., [2]) to specify customized dialogs between user and system and to describe the relevant domain knowledge. The dialog manager is integrated and modeled as a part of the ontology which enables the customization of the dialog management itself based on rules (inspired by [3]). By applying complex dialog rules, the system enhances the built-in dialog capabilities of common VAs. The runtime framework allows to integrate dynamic knowledge from external sources to enrich the user experience. In an automotive scenario the dialog system would e.g. access data from the car (e.g. the fuel level) and from the cloud (e.g. open

gas stations around) and combine this knowledge with dialog history and user context to enable smart human machine dialogs.

VADiMoS abstracts the complex definition of dialog rules by providing a template-based rule editor. This enables the interactive and test-driven creation of dialog rules. The modeler is supported by a guided workflow. Dialogs can be simulated within VADiMoS or directly uploaded to the backend of the voice assistant. The car will connect to the VA and the VA can be configured to communicate with the dialog backend [1]. Alternatively, the runtime modules can be downloaded to supplement also on-board voice engines.

Many automotive OEMs have their own toolchain to implement voice dialogs for state-of-the-art onboard ASR and TTS engines and to connect to the applications like media, telephone, or navigation. To enable off-board dialog development, to extend popular VAs with brand specific dialogs, to be voice assistant independent, and to design on-board dialogs in one tool, a web based tooling will help the OEMs to make the transition easy and comfortable.

In the following section we present our use case. Section 3 will explain how this use case is implemented using our ontology based approach. Section 4 describes how this can be achieved in a comfortable and interactive way using VADiMoS. Finally we will conclude and compare our approach with other tools.

2 Dialog Scenario

In our previous publication we explained complex use cases which require reasoning and anaphora resolution ("how much is it?", [1]). In this paper we focus on a single use case where we augment our knowledge graph with dynamic data that is used for response generation and for further dialog steps. We will showcase how it can be modeled in our ontology and on top we will explain how it is simplified in VADiMoS.

User: “Find open shops nearby?”
System: “REWE is open”
User: “Drive me there.”

A more complex version of this use case would contain disambiguation:
User: “Find open shops nearby?”
System: “There is an Aldi and a Lidl within 500 meters”
User: “Drive me to Lidl.”

This use case comes along with the following challenges: (1) The dialog system in the cloud needs to know the position of the car, (2) supermarkets nearby are dynamic data and need to be added to the knowledge base dynamically on demand, and (3) a list of dynamic data could be part of the response prompt. Finally, (4) the system needs to resolve the anaphora “there” or the user is asked to disambiguate from the list of dynamic data that has been added to the recognizer’s vocabulary.

3 Dialog Approach

In this paper we use a declarative approach to define dialogs as described in [2]. Thus, no coding is required and models are portable and can be re-used independent of the underlying voice assistants. Dialogs and the dialog management are rule based and the rules itself are part of the ontology. The dialogs can run as backend to several voice assistants. Our run-time system contains a dispatcher unit that translates dialog actions into corresponding calls for the assistant. For instance, Alexa provides dialog directives for slot elicitation or confirmation to be used from the Lambda function[1].

[1] https://developer.amazon.com/en-US/docs/alexa/custom-skills/dialog-interface-reference.html#directives

```
{ "@type": "Rule",
  "condition": [{
      "@type": "DialogState",
      "currentIntent": "findShops",
      "isComplete": true
    },{
      "@id": "CurrentCarPosition",
      "longitude": "?carLongitude",
      "latitude": "?carLatitude"
    }
  }],
  "action": [{
   "@type": "HerePlacesQuery",
   "longitude": "{carLongitude}",
   "latitude": "{carLatitude}",
   "radius": "1000",
   "poiCategory": "Shopping",
  }
]}
```

```
{ "@type": "Rule",
  "condition": [{
      "@type": "ExternalResult",
      "data": {
        "@type": "HerePlacesNearby",
        "geoStores": {
          "name": "?name",
          "isOpen": #true,
          "latitude": "?latitude",
          "longitude": "?longitude"
      } }
  }],
  "action": [{
    "@type": "Say",
    "utterance":"{name[n<2]} is open"
  },{
    "@type": "Say",
    "utterance":"{name[n>1;0:n-2]}
       and {name[n-1]} are open"
  },{
    "@type": "UpdateCarSettings",
    "listItems": {
      "name": "destination",
      "value": "({latitude}, {longitude})"
}]}
```

Figure 1 – Rules to handle the request "Find open shops nearby": The left rule checks the *currentIntent* and triggers a cloud service request. The car location is read into variables, e.g. *?carLongitude*, whereas *{carLongitude}* is a placeholder for the retrieved value. The right rule checks for the found places and triggers the *Say* response.

3.1 Ontology

The dialog agent's knowledge and behavior are defined by the system's ontology, a collection of dialog-relevant classes, instances, and their semantic relations. The *static* part of the ontology is defined by the agent's modeler. It consists of the *dialog manager*, the *dialog model* and the *domain model*. The dialog manager defines the structures and rules necessary to determine the "next" dialog move. The dialog model contains the agent-specific language resources such as prompts or sample sentences for speech output and input, as well as dialog-constituting elements like user intents and slots [2]. Finally, the domain model contains classes and objects for knowledge representation and reasoning. The *dynamic* part of the ontology consists of objects which are added to, removed from, or modified in the knowledge base at run-time.

Fig. 1 shows a rule that is triggered after uttering "Find open shops nearby". It checks the existence of the recognized intent *findShops* in the knowledge base and triggers an action. The second rule checks if dynamic knowledge has been created describing shops nearby and if at least one of the shops is open. Dependent on the number of resulting shops it triggers one of the *say* actions, e.g. "REWE is open".

Our ontology format is based on RDF Schema. We decided on using a JSON syntax similar to JSON-LD, because JSON is widely-used in the developer community, easy to understand, and less verbose than XML. A further advantage of the JSON-LD-inspired syntax is the object-centric rather than triple-centric focus [4]. It makes it easier to generalize the underlying RDF knowledge model towards a more object-oriented approach. This enables the dialog designer to abstract or derive any class within the ontology to simplify modeling of standard behavior. Our web interface enables the usage of these abstract classes.

[2] Intents typically relate to a function a user wants the system to execute, for instance, initiating a route guidance. Intent classes have properties, called `slots`, which can be regarded as the parameters of the function an intent represents.

```
{ "@type": "Rule",
  "condition": [{
      "@type": "DialogState",
      "currentIntent": "findShops",
      "isComplete": true
    },{
      "@id": "CurrentCarPosition",
      "longitude": "?carLongitude",
      "latitude": "?carLatitude"
    }, {
      "@type": "ExternalResult",
      "data": {
        "@type": "HerePlacesNearby",
        "position": {
          "longitude": "{carLongitude}",
          "latitude": "{carLatitude}",
          "radius": "1000"
        }
        "geoStores": {
          "name": "?name",
          "isOpen": #true,
          ...
}}] ...
```

Figure 2 – Rule condition to combine both rules from Fig. 1. The cloud service request to retrieve the *geoStores* is opaquely triggered based on the *position*.

3.2 Rule Engine

The dialog agent's "business logic" is modeled with rules. Each rule can be regarded as a condition-action pair. A condition is a set of objects (possibly including placeholders) that must match objects in the knowledge base. The rule's action is executed when all objects match. Rule handling and action execution are performed by our rule engine. It implements a standard forward-chaining production system approach as, for instance, described in [5].

We treat rules the same as other objects in the system as they are also part of the ontology. In particular, property inheritance is also applicable to them. That way, it is possible to create an abstract rule with conditions and actions passed on to derived rule classes. Derivatives may specialize the base rule by adding further conditions and/or actions. This principle is used to avoid repetitions and supports keeping the rule set concise.

3.3 Virtual Knowledge Base

One key factor for convincing automotive dialog systems is that all relevant information is available in a common format. Our system architecture abstracts from the technology used to access a knowledge source. There is an adapter unit for each knowledge source and a central KB manager unit obfuscating whether information comes from the internal knowledge base or some external resource. The dialog modeler does not have to be aware of the information sources. When rules are evaluated at system run-time, knowledge adapters are asked whether they are able to retrieve matching objects for (parts of) the rule condition and/or action. Partial results are joined by the managing unit. This will simplify our example from Fig. 1 by combining both rules into one extended condition part, see Fig. 2.

The first part of the rule condition checks for a certain dialog state, while another part requires a list of shops. The dialog state is checked in the local knowledge base, while the shop objects are opaquely forwarded to a web service. Changing the respective rule in the dialog model is not required, even if the web service is replaced by another service or if shops are resolved internally. What we gain is a portable and extensible dialog model.

4 VADiMoS

The previous sections explained the foundations of our tooling. Our approach provides a huge flexibility and enables the definition of complex dialogs and the usage of external knowledge sources using a JSON-LD inspired definition. With the ongoing development of further intents and the corresponding resource definitions in multiple languages it is hard to keep the overview about the complete project structure. To improve the definitions and ease the handling of the project structure and the dialog rules we created the browser based Voice Assistant Dialog Modeling Service (VADiMoS).

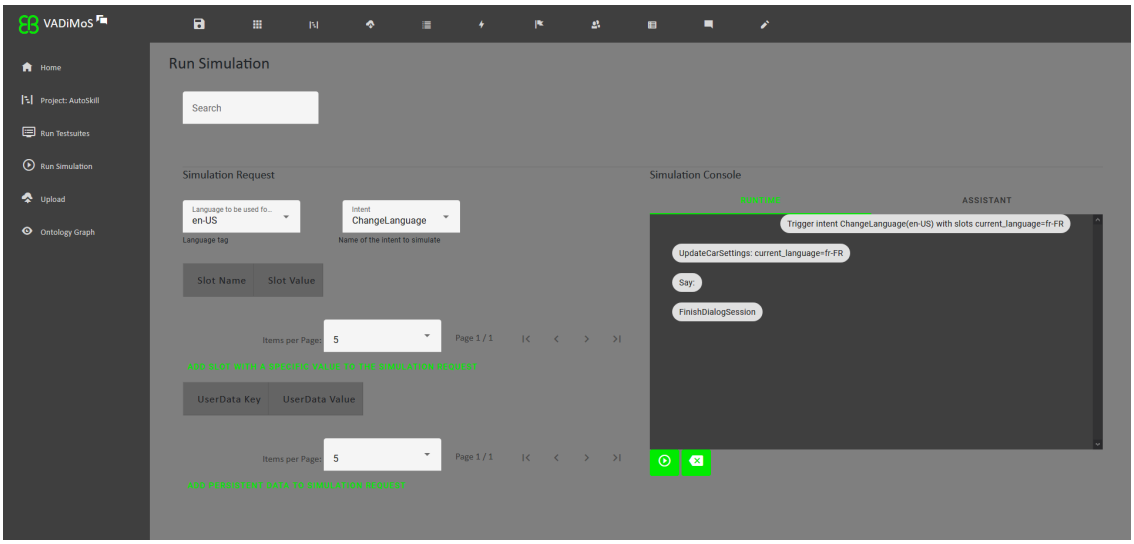

Figure 3 – Simulation of a dialog act in VADiMoS

4.1 Project Management, Testing and Simulation

VADiMoS includes a project manager to maintain multiple projects of an authorized user. It is possible to export single projects to store them in a local version control system. The deployment to the cloud engine and the safe storage of the necessary credentials are managed in an own upload page. It is still possible to download the resulting JSON-LD project structure and upload it manually with the command line tool described in [1]. The command line approach requires the local installation of additional tooling for the common voice assistants. With the web based VADiMoS approach such installation is no longer necessary.

The command line approach already supported the execution of testsuites. With the web based approach VADiMoS introduces a test-driven way to define the testcases before creation of intents and other dialog resources. All components are using the same runtime which is also executed online. This approach enables to test all required dialog acts before uploading to the specific voice assistant eco system. The results of a test run are visualized in a table including the executed testsuites, the corresponding testcases and the corresponding test duration.

Another way of exploring the current dialog capabilities of a project is to run the simulation. The simulation uses the runtime to execute dialog acts on intent and slot basis. Fig. 3 shows the simulation page. The simulation request used for the dialog act is defined on the left side. The user selects the current language, an intent and could configure optional slot values for the intent and user data items, for example the current position of the car. The simulation button on the right bottom side starts the simulation. The simulation console shows the incoming dialog act request from the dispatcher connected to the voice assistant and the dialog action's answer of our runtime. It is possible to simulate single as well as multiple dialog turns including slot elicitation and other dialog actions for example the update of car settings or the car destination.

The usage of external services e.g. to search for open shops is possible by adding plugins and the required credentials in the plugin manager. The plugins provide additional templates and definitions for the domain model. The activated plugins interact with the virtual knowledge base as described in section 3.3.

4.2 Editors

The dialog model and the domain model are defined using dedicated editors. VADiMoS provides an invocation editor, language editor, user data editor, dialog resource editor, intent editor, and a rule editor.

The languages of a project are defined during the project setup. The language editor supports to add or remove languages of a project. The invocation editor is used to define the invocation phrases for each language to activate an action or skill. It ensures the consistency of

the project by requiring the definition of an invocation phrase for each configured language. It is not possible to add a single invocation phrase without translations into the other languages. Intents, dialog resources and user data are grouped in domains.

The user data editor is used to define domain model objects and instances which are exchanged with external plugins or the device (car) connected directly to the runtime or via the voice assistant. The intent editor supports the creation of intents and the definition of slot names as well as the corresponding slot types and values for the domain model. The resource editor is used to create the language specific dialog resources. It helps to create the intent invocation phrases, dialog responses, slot elicitation prompts and slot elicitation responses. The consistency of the resources in multiple languages is ensured by using the same restrictions as the invocation editor.

4.3 Modeling Rules

With the help of the rule editor the modeler is able to create the dialog logic used to power the rule engine described in section 3.2. A visual node editor can be opened for each intent. Multiple nodes are interactively connected to create one or multiple dialog rules for an intent. The node editor is organized in node categories. It supports input nodes, filter nodes, condition nodes, and action nodes created from abstract rule templates which are defined in the static part of the ontology mainly in the dialog manager definition. Plugins are providing templates which introduce and add additional nodes to the node categories. The input nodes category contains for example an intent node, slot node, and an user data node. The condition nodes correspond to the rule templates, for example to check for an intent with a specific name and trigger the response action. A basic workflow could be described as the following: The modeler creates an input node, e.g. *Intent*, and connects it to a condition node. The condition is requires an intent node as input and an action node, e.g. *Say*, as output . The condition checks for example for a specific intent name. A *Say* node is connected to the condition as an output action. It is used to configure the dialog resource needed for the response prompt. Plugins use a common plugin interface in our ontology. The nodes which are added by a plugin are connected by offering an *ExternalAction* input or an *ExternalResult* output. The results of a plugin can be filtered using an *FilterExternalResult* node from the filter node category. To support the modeler not require to remember certain user data, slots and other dialog resources, the editor offers the possible values and objects which could be used for the specific nodes.

The use case which we described in section 2 and the rule definition which we showed in Fig. 1 is modeled in the rule editor and shown in Fig. 4. The first rule on the top is checks for the intent name and triggers an external action to search for the open shops. The query action is described by the plugin template interface. The results of the query can be limited and the radius for the search can be specified. The query uses the user data to execute the action using the current car position. If the query action is finished an *ExternalResult* is received and another rule is used to filter the results of the plugin. All objects contained in the *ExternalResult* are automatically displayed and accessible in the filter as soon as we connect the filter with the *ExternalResult*. In our case the filter is restricting the result to all open stores and transfers the filtered result to the condition node. The condition node is a generic node which is dynamically receiving the domain objects it might check and use for updating certain car user data. In this example the condition node is resolving the name of the shop, the destination latitude and longitude, and uses them for the response *Say* action and for the update of the car destination with the *UpdateCarSettings* actions.

4.4 Visualization of the Ontology

The ontology visualization could be used to obtain knowledge of specific domain objects which are available in the project. The filter node in Fig. 4 shows a list of *geoStore* objects contained in the *ExternalResult*. We could search for the term “Geo” in the search field of the ontology visualizer, shown in Fig. 5, to gather more information about this ontology object and its class

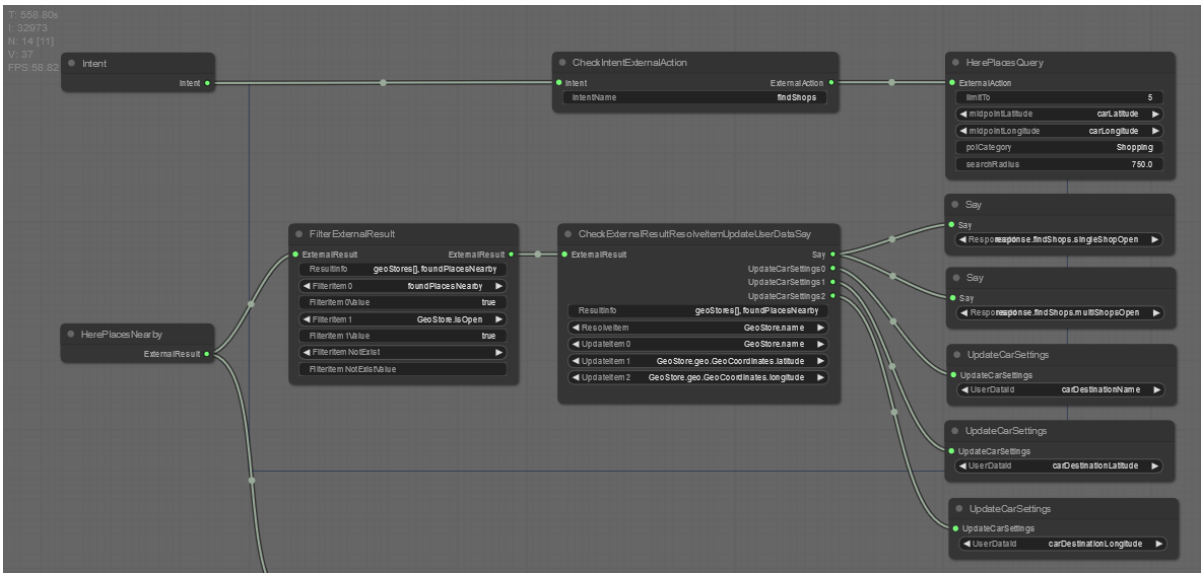

Figure 4 – findShop intent modeled in the VADiMoS rule editor

structure. *geoStore* is of type *PointOfInterest* which has a member *GeoCoordinates*. *GeoCoordinates* has properties for the position, in our case the position of the car. The interactive exploration of the ontology graph enables the lookup of complex ontology domain objects which are for example derived from multiple sub classes. The visualization supports the creation of rules and finally gives a deep look inside of the specified ontology.

5 Conclusion and Outlook

Most of the common voice assistant eco systems require to program the handling of complex dialogs manually by writing source code or provide a rather simple way of dialog handling and the dialog management is used from the eco system as a black box. In our approach we use a common ontology which enables the exchange of information in a generic way. The description of the dialog management is part of the ontology, the test-driven and interactive editing and the visualization of the complex underlying ontology enables the creation of complex and satisfying dialogs for users. Our approach is supporting multiple voice assistants and is also not limited to the cloud. In the future we want to add support to import ontologies specified by schema.org [6].

Another focus of our future features is neuronal network based dialog management. The RASA framework for example offers different ways to specify or drive dialog management. It is possible to train neuronal networks to produce probabilities which dialog action should be taken (stories[3]) or to specify custom rules[4]. We would like to explore the combination of the rule based approach together with a neuronal network based dialog management. The runtime engine of our system is not only usable for the cloud based voice assistants. With the release of the Alexa Custom Assistant [7] and the Local Voice Control extension for the Alexa Auto SDK [8] we expect further possibilities to use our approach even in the local embedded device (car). The integration of Alexa into the car is done using the Alexa Auto SDK. To support the integration of cloud and local assistants we use the EB Voice Assistant Broker which also supports the communication with our on-device or cloud runtime on the embedded device. It can act as a supervisor of multiple voice assistants and provides a common interface to HMI and the applications.

[3]https://rasa.com/docs/rasa/stories
[4]https://rasa.com/docs/rasa/rules

Figure 5 – Visualization of classes related to search term "Geo" defined in the ontology

References

[1] SAHOO, S. K., T. SOWA, C. HACKER, and T. RANZENBERGER: *Rule-based dialog management for voice assistents in automotive environments*. In R. T. BÖCK, I. SIEGERT, and A. WENDEMUTH (eds.), *Studientexte zur Sprachkommunikation: Elektronische Sprachsignalverarbeitung 2020*, pp. 185–192. 2020.

[2] MOTIK, B., P. F. PATEL-SCHNEIDER, and B. PARSIA: *Owl 2 web ontology language*. W3C, 2012. URL `https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/`.

[3] TRAUM, D. and S. LARSSON: *The information state update approach to dialogue management*. In J. VAN KUPPEVELT and R. SMITH (eds.), *Current and new Directions in Discourse and Dialogue*, pp. 325–353. 2003.

[4] LANTHALER, M. and C. GÜTL: *On using JSON-LD to create evolvable RESTful services*. In *Proceedings of the Third International Workshop on RESTful Design*, WS-REST '12, pp. 25–32. Association for Computing Machinery, New York, NY, USA, 2012.

[5] RUSSELL, S. and P. NORVIG: *Artificial Intelligence: A Modern Approach*, chap. 9.3. Prentice Hall Press, USA, 3rd edn., 2009.

[6] *Home - schema.org*. 2021. URL `https://schema.org`. Accessed 21.01.2021 18:00.

[7] *Amazon announces alexa custom assistant*. 2021. URL `https://developer.amazon.com/en-US/blogs/alexa/alexa-auto/2021/01/Amazon-Announces-Alexa-Custom-Assistant`. Accessed 21.01.2021 18:00.

[8] *Accelerate the integration of alexa into your vehicles*. 2021. URL `https://developer.amazon.com/en-US/blogs/alexa/alexa-auto/2020/10/Accelerate_the_integration_of_Alexa_into_your_vehicles`. Accessed 21.01.2021 18:00.