RULE-BASED DIALOG MANAGEMENT FOR VOICE ASSISTANTS IN AUTOMOTIVE ENVIRONMENTS

Soyuj Kumar Sahoo, Timo Sowa, Christian Hacker, Thomas Ranzenberger

Elektrobit Automotive GmbH
soyujkumar.sahoo@elektrobit.com

Abstract: We describe a rule-based voice dialog system for in-vehicle infotainment. The system is 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. We introduce the concept of a virtual knowledge base to accommodate the diversity of dialog-relevant knowledge sources in a car. It unifies the knowledge access for the dialog logic, no matter whether an internal knowledge base or an external source like a web service is being used. The virtual KB thus supports the model's portability and re-usability. Dialog models can be exported and deployed to on-market voice assistants like Amazon Alexa and Google Assistant, which essentially means to extend their built-in dialog management capabilities. The dialog management run-time system is deployed on a separate cloud server and acts as a back-end for the respective voice assistant.

1 Introduction

Like Rome, artificial intelligence (AI) cannot be developed over night. With the rise of voice assistants (VA), this decade can certainly be dedicated to solving one of the goals of AI towards the realization of natural language understanding and dialog between humans and machines. VAs such as Amazon Alexa, Google Assistant, and Siri, have started to become popular, especially when the usage of touch interfaces is inconvenient or even cumbersome to interact with, for instance while cooking, exercising, or driving.

However, to develop an in-vehicle dialog system using on-market VAs is still a big challenge. Out-of-the-box they only support a limited set of dialog phenomena and no possibilities to integrate various knowledge sources. Modern cars have access to multiple, very different, sources of knowledge. There is dynamic car-internal data like velocity, fuel/charge level, exterior/interior sensors, as well as Car-to-X communication, connected mobile devices, open and personalized cloud-based services such as REST APIs for places, Wikidata[1], or personal contacts in the web. There is also static car-related data like outer dimensions, engine type, interior, fuel tank/battery capacity, all of which can play a role in dialog. Common ontologies with a broad [1] or more automotive-specific [2, 3] focus provide the means to unify this data and make it available for domain-spanning reasoning and smart dialog systems[2].

In this work we provide a solution to handle complex dialogs together with external knowledge access. The proposed system provides an intelligent dialog management on top of the existing VAs, to enrich the in-car voice user experience. Our dialog management approach uses

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

[2] cf. also https://www.w3.org/community/gao/

an ontology [4] to specify the interaction between user and system. It allows the user to define the dialog model and the dialog manager itself in a declarative and rule-based way (inspired by [5]) that does not require any implementation. In contrast to VAs, that solely focus on dialog specification, our system additionally provides the means to define how dialog connects to the domain model and how to integrate arbitrary knowledge into this model.

In the following section we describe an exemplary in-car dialog scenario. Section 3 describes our approach towards ontology- and rule-based dialog management and reasoning. Section 4 provides examples how to model the described scenario in our ontology and how the system behaves in these examples.

2 Dialog scenario

A good dialog manager needs to deal with various use cases. We use an navigation scenario to illustrate different challenges regarding dialog phenomena and access to external knowledge sources. In a first step, the user wants to initiate a route guidance. It is possible to navigate to a POI, an address consisting of city, street, and house number, or to a person described by its name and the address type (i.e private or business). Each of these three groups defines a mandatory and sufficient set of parameters that the system needs to elicit before considering the route guidance intent of the user as complete.

U: "Navigate to Thomas."
S: "Thomas, ok. To private or work address?"
U: "Work address."
S: "Ok, navigating to Elektrobit in Erlangen."

The user can specify slots in any order. Each slot is confirmed in an implicit manner. By inferring knowledge available from the car (fuel status, average fuel consumption, distance to destination) and the user's preference (ABC gas stations), the system initiates the following dialog and searches for a suitable result by accessing knowledge sources in the cloud.

S: "You need to refuel on the trip! Driving distance to Elektrobit is 100 km, but your fuel lasts only for 50."
U: "Thanks. Find a place to refuel."
S: "The cheapest ABC filling station along the route is 23 km away."

The system searches along the route, when guidance is active. Now the user can request further information using the anaphoric references "it" (the fuel) and "there" (the last location we talked about). The system can resolve this by storing the dialog context appropriately. Additionally the system takes into consideration that the car uses Diesel.

U: "How much is it?"
S: "Diesel price is 1.23 Euro there."
U: "Ok, drive me there."
S: "Alright, I'm adding the ABC gas station as route stopover."

Inference of car knowledge, user preferences, and external knowledge together with dialog strategies like natural slot filling or dialog history result in short and constructive dialogs.

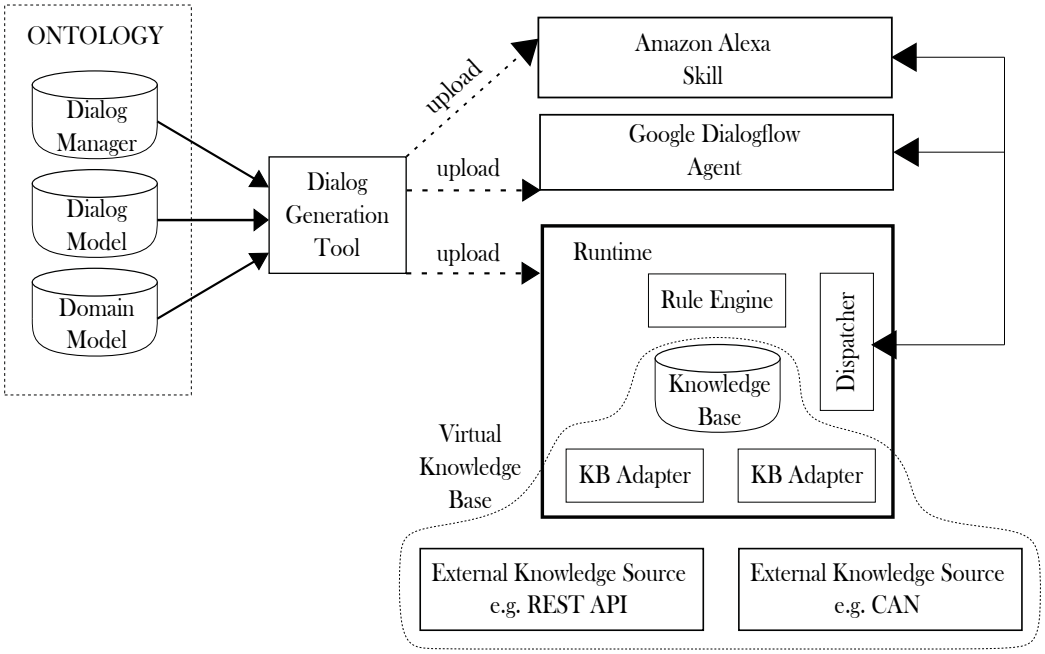

Figure 1 – Overview of the architecture: The ontology is converted to a knowledge base and uploaded by the Dialog Generation Tool. One part is deployed to Alexa and/or Google to enable speech recognition and semantic analysis. The other part is the knowledge base used by the rule engine. It is uploaded to the server hosting the runtime environment/dialog back-end.

3 Approach

Fig. 1 shows an overview of our system architecture. The left-hand side depicts the tooling part consisting of the ontology and the dialog generation tool (DGT). The latter validates, exports, and uploads resources to VA platforms in the web, and to a runtime environment which is currently an AWS λ service.[3] In the following, we will address the depicted components and resources in more detail.

3.1 Ontology

The dialog agent's knowledge and behavior is defined by the system 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. Finally, the domain model contains classes and instances for knowledge representation and reasoning. An automotive dialog agent, for instance, may have a representation of a route guidance system or a gas station (and its relation to fuel) as a part of its domain model. The *dynamic* part of the ontology consists of objects which are added to, removed from, or modified in the knowledge base at run-time.

Our ontology format is based on RDF Schema. It provides the necessary vocabulary to express fundamental relations such as superclass/subclass, or the applicability of properties to certain classes using the domain/range meta-properties. It was considered as powerful enough

[3] https://aws.amazon.com/lambda/

```
"classes": [{
  "@id": "Intent",
  "@properties": [{
      "@id": "slots",
      "@range": "Slot"
    }]
}, {
  "@id": "Slot",
  "@subClassOf": "NamedObject",
  "@properties": [{
    "@id": "typeName",
    "@range": "@String"
  }, {
    "@id": "optionalSlot",
    "@range": "@Boolean"
  }]
}]
```

```
"objects": [{
  "@id": "DriveMeToIntent",
  "@type": "Intent",
  "slots": [{
    "name": "personName",
    "typeName": "PersonType"
  }, {
    "name": "addressMode",
    "typeName": "AddressType"
  }, {
    "name": "city",
    "typeName": "CityType"
  }, {
    "name": "Street",
    "typeName": "StreetType"
  }]
}]
```

Figure 2 – JSON definition of an "Intent" class (left) and instantiation of an intent (right).

for dialog modeling, yet (unlike OWL) still simple enough to be used by modelers without a specific knowledge engineering background. 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 the "standard" XML format. A further advantage of the JSON-LD-inspired syntax is the object-centric rather than triple-centric focus of the syntax [6]. It makes it easier to generalize the underlying RDF knowledge model towards a more object-oriented approach. VAs such as Google Assistant or Amazon Alexa eventually use JSON as specification language for their dialog models, too. Fig. 2 shows an exemplary definition of the classes `Intent` and `Slot`. 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. In the example the `Slot` class is derived from `NamedObject` and inherits the `name` property (not shown). On the right hand side of the figure an instance of the `Intent` class is shown.

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 [7]. The rule engine is applied in two places: First, when running the DGT during export, inference rules are applied to the static ontology to add (make explicit) certain object properties. RDFS entailment rules[4] and property inheritance rules are applied in this step. Inheritance allows to attach properties to a class which are propagated to sub-classes and to all instances. This can be used, for instance, to define default values of a class property such as "all dogs have four legs".

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. Fig. 3 shows an example of rule inheritance.

[4] https://www.w3.org/TR/rdf11-mt/#rdfs-entailment

```
"classes": [{
  "@id": "IntentResponseRule",
  "@subClassOf": "Rule",
  "condition": [{
      "@type": "DialogState",
      "currentIntent": "?intent",
      "isComplete": true
    }, {
      "@type": "CurrentRule"
      "responsePrompt": "?responsePrompt"
      "intent": "?intent"
  }],
  "action": {
    "@type": "Say",
    "utterance": "?responsePrompt"
  }},
{ "@id": "Rule",
  "@properties": [{
      "@id": "condition"
    }, {
      "@id": "action"
  }],
  "@inheritProperty": [
    "condition",
    "action"
  ]
}],
"objects": [{
  "@type": "IntentResponseRule",
  "intent": "DriveMeToIntent",
  "responsePrompt": "Starting guidance."
}]
```

Figure 3 – Classes describing a specialized rule `IntentResponseRule` derived from `Rule` together with one instance (object).

3.3 Runtime for dialog processing

The dialog manager is implemented as a set of processing rules and thus can be regarded as a part of the dialog agent's model. We follow the information state update approach as introduced in [5] by making dialog processing explicit with rules operating on a likewise explicit formal representation of the informational components. The practical advantages for automotive VAs are that potential users/customers can choose to adopt our dialog management approach, modify it, or create a new one without altering any code. Declarative, rule-based dialog management models are portable and can be re-used for different agents or for a different underlying engine (e.g., by exporting them to a W3C standard such as RIF[5]).

When running in the cloud, 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[6]. The rule engine calls the dispatcher when a rule containing a dialog action is executed, which in turn issues the appropriate directive to Alexa.

3.4 Virtual knowledge base

One key factor for convincing automotive dialog systems is that all relevant information is available in a common format. Then the dialog model can link up knowledge from different areas, broadening the possibilities for a good user experience. For the information state approach (IS), the common, explicit interface is emphasized as follows:

> Other resources . . . can also be integrated in the system, using an interface that allows the same kinds of queries and operations as for the IS proper, allowing update rules to be oblivious as to whether the components are part of the information state or external resource. [5], p. 14.

Our system architecture abstracts from the technology used to access a knowledge source. There is an adapter unit for each knowledge source ("KB Adapter" in Fig. 1) and a KB manager unit obfuscating whether information comes from the internal knowledge base or some external

[5]https://www.w3.org/TR/rif-overview/

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

resource. Our demo system currently implements access to web services for location-based information such as route planning, and a service to request real-time fuel prices.

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. For instance, one part of a rule condition may check for a certain dialog state, e.g., that the current intent is to drive to some address, while another part requires the address to have certain geo coordinates. The dialog state is checked in the local knowledge base, while the location-related object may be opaquely forwarded to a geocoding web service. Changing the respective rule in the dialog model is not required, even if the geocoder is replaced by another service or resolved internally. What we gain is a portable and extensible dialog model.

3.5 Deploy to multiple VAs

The Dialog Generation Tool (DGT) is currently able to export dialog models for Amazon Alexa skills and Google Dialogflow agents. Both Alexa and Dialogflow are used as NLU (natural language understanding) engines, but the dialog management is completely driven by our run-time system in the cloud. The ontology model for both engines is identical and DGT takes care about special handling for built-in intents and slot/entity types. Though we currently deploy the run-time system to the cloud, is would also be possible to export it for a local (embedded) engine, and run it in the car.

4 Modeling examples

In the following we want to look at different aspects from the use case presented in Section 2. The core element to model such a dialog behavior is the `Rule` as described in Section 3.2. A rule consists of a condition and a list of actions which are executed if the condition matches. Whenever the condition does not match, other rules are executed. Efficient evaluation of rules can be done with the Rete algorithm [8].

The condition part of each rule in Fig. 4 shows a conjunction of predicates. A predicate is true, if a matching instance exists in the knowledge base. In the right column you find e.g. a predicate that checks whether the knowledge base contains an object of type `DialogState` with properties `currentIntent` and `isComplete` and that the properties have the respective property values, i.e. `PriceIntent` and `true`. This means, that the predicate evaluates to true, if the semantic analysis of the user's utterance by the VA has recognized the `PriceIntent` and if other rules of the dialog manager already detected that this intent is complete and no further slots need to be elicited. VA and dialog manager must have already asserted these facts in the knowledge base.

Some predicates have properties with a variable instead of a value: In the left column of Fig. 4 we check all objects of type `Route` and assign the value of its property `distance` to the variable `?destination_dist`. This syntax is similar to SPARQL [9] which is used internally as a query language for knowledge requests. The example is simplified since we assume that there is only one `Route` object in the database; thus the variable contains only one distance value. In a similar way `?drive_dist` is requested from the `CarConfig` i.e. from the configuration data that is synchronized from the car manufacturers cloud or directly from the car environment. The predicate `IsGreaterThan` compares both values. The syntax `{drive_dist}` implies that the interpreter reads this variable; all predicates containing `?drive_dist` (writing to the variable) need to be evaluated before.

The rule in the left example has two actions. One action causes the prompt *"You need to*

```
{ "@type": "Rule",
  "condition": [{
      "@type": "Route",
      "distance": "?destination_dist"
    }, {
      "@type": "CarConfig",
      "fuel": "?fuel",
      "drivedistance": "?drive_dist"
    }, {
      "@type": "IsGreaterThan",
      "value1": "{destination_dist}",
      "value2": "{drive_dist}"
  }],
  "action": [{
      "@type": "Say",
      "utterance": {
        "value": "You need to refuel."
    }, {
      "@type": "KnowledgeAssert",
      "fact": {
        "@id": "DialogHistory",
        "objects": {
          "@type": "Product",
          "name" : "{fuel}"
        }
      }
}]}
```

```
{ "@type": "Rule",
  "condition": [{
      "@type": "DialogState",
      "currentIntent": "PriceIntent",
      "isComplete": true
    }, {
      "@id": "DialogHistory",
      "objects": {
        "@type": "Product",
        "name" : "?prod" }
    }, {
      "@id": "DialogHistory",
      "objects": {
        "@type": "Organization",
        "name" : "?store" }
    }, {
      "@type": "Organization",
      "name" : "{store}",
      "makesOffer": {
        "price": "?price",
        "itemOffered": "{prod}" }
  }],
  "action": [{
    "@type": "Say",
    "utterance":"{prod} price is {price}"
}]}
```

Figure 4 – JSON definition of a "Rule" initiating a pro-active prompt to refuel (left) and a response to the request *How much is it?* (right).

refuel" to be sent to the skill or agent via the dispatcher unit (Fig. 1). The `KnowledgeAssert` action modifies the knowledge base: it changes the object with ID `DialogHistory` and replaces the referenced object of type `Product`. The value of the product is then `{fuel}`, i.e. the content of the variable which has been read before from the car configuration during condition evaluation. The system talks about the product "fuel" which is "Diesel" in the current environment available form the car configuration. We need to remember this for the later evaluation of the rule on the right side.

Fig. 4, right, shows the rule which evaluates to true after the user requests *"How much is it?"*. The `currentIntent` has been set to `PriceIntent` and from the `DialogHistory` we know the name of the product we talked about: the variable `?prod` will hold the value "Diesel". With the utterance *"The cheapest ABC filling station . . . "* in our scenario, the system already introduced a concrete filling station into the dialog context, resulting in an object of type `GasStation` to be added to our dialog history. The name of the gas station is retrieved from the dialog history and stored into the variable `?store`. Note that `GasStation` inherits from `Organization` which has a property `makesOffer` like defined in [1]. The rule evaluates to true, since the product we talked about matches to the offering of the gas station. The price of the offering is written to the variable `?price`, because it is retrieved from a web service for real-time fuel prices. The corresponding adapter accepts queries about organizations of type `GasStation` and returns the current prices for several fuel products.

The rule on the right side has only one action which initiates the prompt *"Diesel price is 1.23 Euro"*. The rule is generic: It evaluates to true and returns the current prices if any organization and any matching product is available in the dialog history and if there is an adapter registered that can deal with these types.

The rules in Fig. 4 can be simplified, when we extract common used patterns to generic dialog manager rules. A concrete rule for an intent does not need to take care about objects in

dialog history. The same holds for natural slot filling: elicitation of missing slots and confirmation is handled with generic dialog manager rules and finally result in a modified dialog state by setting `isComplete` to true. The dialog strategy itself can be easily modified by adapting rules but no code.

The final model that allows to add an address, finds the right filling station if necessary and gives information about the prices can be easily verified by executing JSON based test cases or by uploading the model to the runtime environment (see Fig. 1).

5 Conclusion and outlook

We demonstrated how dialog definition, management, and domain information can be packaged in a uniform ontology-based JSON model that fuels a dialog system. The approach abstracts from knowledge access and supports creating portable and re-usable models for intelligent dialog systems. Our next steps will include creating adapters for dynamic vehicle data such that sensor information is accessible in the dialog. This includes extending the rule engine by adding support for highly dynamic data, which is ubiquitous in the automotive domain. Classical rule-based reasoning assumes static or at most slowly changing knowledge, instead.

References

[1] *Home - schema.org*. 2020. URL `https://schema.org`. Accessed 21.01.2020 18:00.

[2] FELD, M. and C. MÜLLER: *The automotive ontology: Managing knowledge inside the vehicle and sharing it between cars*. In *Proceedings of the 3rd International Conference on Automotive User Interfaces and Interactive Vehicular Applications*, AutomotiveUI '11, p. 79–86. Association for Computing Machinery, New York, NY, USA, 2011.

[3] KLOTZ, B., R. TRONCY, D. WILMS, and C. BONNET: *VSSo: The vehicle signal and attribute ontology*. In M. LEFRANÇOIS, R. GARCÍA-CASTRO, A. GYRARD, and K. TAYLOR (eds.), *Proceedings of the 9th International Semantic Sensor Networks Workshop, International Semantic Web Conference*, vol. 2213 of *CEUR Workshop Proceedings*, pp. 56–63. 2018.

[4] 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/`.

[5] 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.

[6] 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.

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

[8] FORGY, C.: *A fast algorithm for the many pattern/many object pattern match problem*. *Artificial Intelligence*, 19(3), pp. 17–37, 1982.

[9] GARLIK, S. H. and A. SEABORNE: *Sparql 1.1 query language*. W3C, 2013. URL `https://www.w3.org/TR/2013/REC-sparql11-query-20130321/`.