A Toolkit for Nested Multi-turn Speech Dialog in Automotive Environments

Timo Sowa, Soyuj Kumar Sahoo

Elektrobit Automotive GmbH
timo.sowa@elektrobit.com

Abstract: Current in-car dialog systems foster concise interactions for navigation, media, or phone control very well. Yet, they are usually not employed for complex multi-turn dialogs involving nested topics. However, this type of conversation is getting more important when moving toward more natural, assistant-like voice interfaces in the car. We present a dialog modeling and execution approach to account for this. It is based on the idea of a dialog stack that keeps track of the active dialog topics. The stack influences the execution of dialog turns while executed dialog turns modify the stack. The approach supports dialog decomposition into reusable parts and topic interruption/resumption for user-initiative and mixed-initiative dialogs. We implemented our approach for an existing GUI and dialog modeling tool.

1 Introduction

Voice user interfaces are nowadays very common in cars. For almost any car model there is a variant of the infotainment system that includes voice control. According to a German survey, about 25% of the drivers of "voice-enabled cars" are using this feature. Typical use cases for in-car voice interaction include navigation, radio/media control, and telephony (though recent infotainment systems cover many more).

User utterances are mapped to (parametrized) functions of the infotainment system during speech processing. The processing chain normally consist of speech recognition, semantic interpretation (extracting the function and its parameters from the utterance), and dialog management. The latter coordinates speech recognition and output. The infotainment functions (also called *intents*) are usually triggered by voice commands that contain the required parameters (also called *slots*). For instance, a phone command like "call Peter at work" may contain a slot for the callee (Peter) and a slot for the contact type (at work). For typical use cases, it's often possible to trigger a function with a so-called *one-shot* command which specifies all the required slot values in a single utterance. The more slots a certain function requires, however, the more difficult it gets to express all of them in a one-shot command. Hence, the system should ideally be able to cope with different interaction styles including one-shot and prolonged *multi-turn* dialogs. When a dialog stretches over several turns, it should be interruptible and resumable to increase the efficiency of the interaction (avoiding to start over after the interruption). Such interruptions (or deviations from the dialog topic) may come from the user him/herself (e.g. when asking for the weather in the middle of a more complex interaction), or from the system which may take the initiative and inform the driver about important traffic hazards or dangers on the route while he/she is engaged in voice interaction.

Dialog design tools should provide an appropriate support for modeling such features. Efficient dialog modeling also means not to reinvent the wheel. In particular, when designing a voice interface for a multitude of different functions, there may be dialog patterns that can be reused.

Fueled by the recent success of voice assistants, users' expectations toward in-car voice interfaces may rise. Voice interaction needs to be more natural and needs to cover more complex functions, also beyond the immediate car-related use cases. Dialog modeling and execution approaches as well as the tooling need to take this into account.

We present a dialog specification and execution approach that aims to support these aspects. It is an extension of an existing product for in-car HMIs (human-machine interfaces) including voice user interfaces. In the following section we will briefly describe the dialog management/modeling approach we follow in our existing product. Section 3 describes our extended approach focusing on the mechanics of dialog stack management. Notes on the prototype system and concluding remarks can be found in the final sections.

2 State-based dialog design and management

The general HMI modeling approach we follow in our product[1] is based on Harel state graphs [1]. An HMI may consist of multiple state graphs which are processed in parallel by state machines. The synchronization between different state machines is achieved with events that can be received and processed in all state machines. Graphic/haptic and speech interaction are modeled with separate graphs. The state graph for speech consists of *talk states* that define voice interaction and of transitions between the states. Talk states in turn contain elements that constitute the dialog (also called *spidgets*, "speech widgets"). Prompt elements specify system utterances, while command elements define the inventory of recognizable utterances (via grammars or example sentences) and the behavior upon recognition [2].

Each dialog element is configured with a set of *properties*. Properties are named local variables that can be set by the modeler and may change during run time. A command spidget for speech input, for instance, has properties to define the utterances to recognize. Further properties define what to do when a command was recognized. A typical reaction would be to fire an event causing a transition to another talk state. When a talk state is entered at runtime, the standard system behavior is to play the prompts defined in that state, followed by speech recognition as defined by the commands. With this standard approach, the dialog's structure is mainly defined by the state graph. This modeling method is easily comprehensible, but has limitations when it comes to complex dialogs with an adaptive dialog flow as described in the introduction. For this reason we extended our approach as described in the following sections.

3 Extended dialog modeling approach

3.1 Ontology, intents, and entities

We introduced an ontology to be able to express structured information both regarding the intents and the data they're dealing with. It is a resource coupled with an HMI model. With the ontology a modeler can freely define classes of objects using inheritance and part-of relations. One built-in class is called `Intent` and serves as a superclass for model-specific dialog topics or functions. Each class derived from `Intent` typically relates to a function a user wants the system to execute. Intent classes have members, which can be regarded as the parameters of the function an intent represents. Members are themselves defined in the ontology as derivatives from the built-in `Entity` class. The modeler may assign properties of dialog elements (prompts, commands) to intent types. This is used to control the activation of dialog elements and to define their effect (see below).

[1]EB GUIDE + Speech Extension.

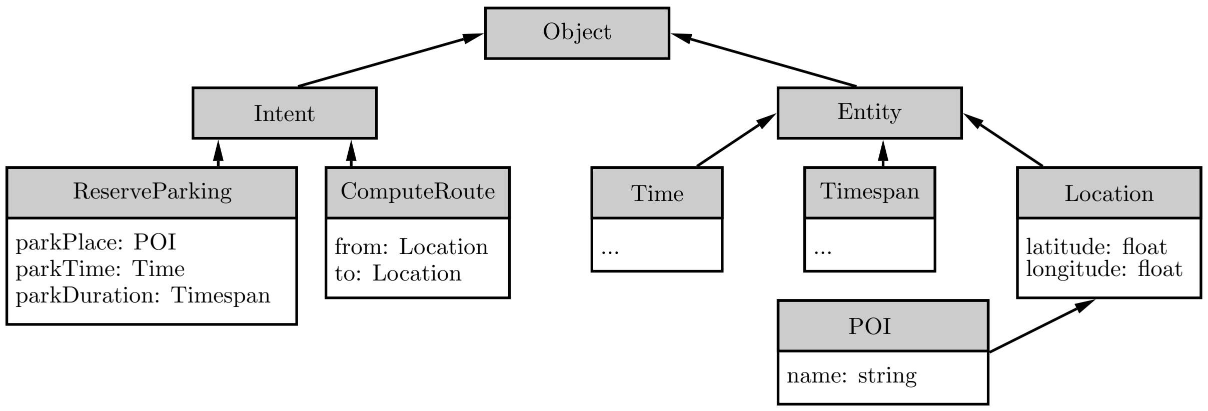

Figure 1 – A simple ontology defining intents and entities. Boxes depict classes and show their members, arrows show is-a relations.

3.2 Combined state-based and information state update approach

We further extended our approach by combining it with modeling capabilities more aligned to the information state (IS) update approach [3]. We keep the state graph, but modify the meaning of a (talk) state to just define the set of available dialog elements. The elements may be active or inactive depending on the information state. The modeler can define the conditions for activation for each dialog element (prompts and commands) independently. Additionally, he/she defines how to alter the information state when the respective dialog elements get executed.

The processing cycle of a talk state is illustrated in Figure 2. A talk state is entered due to an event, and the activations for all prompts in the set are evaluated. The system executes (plays) the active prompts. Afterwards, the information state is altered according to the updated properties for the IS of the active prompts. The updated information state is then applied to evaluate the activation for all commands in the set. Speech recognition is initiated if there is at least one active command. In case an active command has been recognized, the information state gets updated according to its update properties. This change may activate other prompts. The processing cycle continues until there is no more active dialog element. That way, a complex dialog may unfold within a single state of the HMI model. Making dialog elements depend on the IS, however, is optional. If the modeler does not use the feature, the behavior falls back to the pure state-graph-driven approach described above where simply all prompts and commands are activated. By embedding the IS-based execution of dialog elements in a state, we still allow the limited, but easy and straightforward state-based modeling approach.

3.3 Call back questions and confirmations

One requirement mentioned in the introduction was to support flexible, multi-turn dialogs. While they can be modeled with the state-based approach, state graphs tend to get complicated when trying to insert some flexibility. For instance, if some slots are filled with an initial

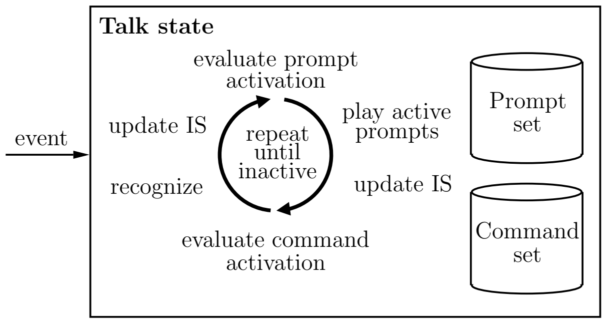

Figure 2 – Processing cycle when the state machine enters a talk state.

command, and the system shall flexibly ask for missing slot values and confirm them. This kind of dialog strategy is often called *slot-filling* or *frame-based* approach. It requires the system to keep track of the currently active intent, the values of its members, and whether the values have been confirmed.

In our framework it can be achieved by setting dedicated properties. The property `preIntent` is available for any dialog element. It is set to an intent class defined in the ontology. When present, the respective dialog element is only considered for activation if the value of `preIntent` matches to the current intent (or topic under discussion). Additionally, there is the `postIntent` property which sets the current intent after the dialog element has been executed. To enable slot-dependent activation the property `member` can be used. All members defined for `preIntent` are selectable as value. For specifying call back questions and activating subsequent commands, the binary-valued `undefined` property can be used. If set to true, it is required that the slot value `member` of the intent `preIntent` is undefined (thus, has no value yet). Similarly, setting the `confirm` property for prompts requires that the slot was not yet (implicitly) confirmed in order to activate the prompt.

The following example illustrates how to use these properties to model a slot-filling dialog. Assume that these dialog elements are defined in a talk state:

1. **command** to initiate a parking reservation request
   `preIntent = ⊥, postIntent = ReserveParking`

2. **prompt** to request the parking place (POI) from the user
   `preIntent = ReserveParking`
   `member = parkPlace, undefined = true`

3. **command** to recognize a parking place (POI)
   `preIntent = ReserveParking`
   `member = parkPlace, undefined = true`

4. **prompt** as a confirmation for the POI to park at
   `preIntent = ReserveParking`
   `member = parkPlace, confirm = true`

When entering the talk, only the command (1) is activated. The initial user utterance could be “find a parking lot” which is recognized by (1) and causes the current intent to be set to `ReserveParking`. In the next processing cycle prompt (2) is played (e.g. “Where do you want to park?”) and command (3) is activated for recognition since the intents match and the conditions for the member `parkPlace` are fulfilled. The user may say “Close to Shedd Aquarium” which is stored as value for the currently active intent and activates prompt (4) which may confirm the slot by saying “Ok. Shedd Aquarium”. Further dialog elements could be added for the other members of the parking reservation intent.

3.4 Dialog stack

Call back questions and confirmations are basic ingredients for multi-turn capable dialog models. In a similar way, these approaches can be found in the dialog specifications for popular voice assistants such as Amazon Alexa and Google Assistant. For nested dialogs, however, we extended our approach by employing a *dialog stack* model inspired by [4]. The elements of the stack are structures describing all currently active intents. Each element is a triple (i,m,r) where i is an instance of an intent class as defined in the ontology, $m =$

$i = \textbf{WeatherInfo}$
$m = \{(city, Chicago, false), (time, \perp, false)\}$
$r = \{\}$

$i = \textbf{EnterTime}$
$m = \{(time, \perp, false)\}$
$r = \{(time, parkTime)\}$

$i = \textbf{ReserveParking}$
$m = \{(parkPlace, SheddAquarium, true),$
$(parkTime, \perp, false),$
$(parkDuration, \perp, false)\}$
$r = \{\}$

User: "Please reserve a parking lot close to the Shedd Aquarium."

System: "Shedd Aquarium, ok. At what time?"

User: "How is the weather in Chicago?"

Figure 3 – Status of the dialog stack after the discourse shown right.

$(member_1, value_1, confirmed_1), \ldots, (member_n, value_n, confirmed_n)$ defines the current member values for the intent instance *i* together with a flag whether the value was already confirmed or not. $r = (member_{i1}, member_{j1}), \ldots, (member_{in}, member_{jn})$ represents the member values to propagate to the next-lower level with an intent *j* when *i* gets popped from the stack. This can be employed to model sub-dialogs (see below).

Initially the dialog stack is empty. Pushing an element on the empty stack means that a new discourse segment with a dedicated purpose begun. This purpose is represented by the corresponding intent *i*. Pushing an element on the non-empty stack starts a new, embedded discourse segment. Popping an element from the stack means that the discourse segment connected to *i* is finished. This may happen either because all required members of *i* have values and the intent can thus be executed, or because the user explicitly canceled the dialog.

Consider the example shown in Figure 3. The left side of the image is a snapshot of the stack's status after the dialog turns on the right side were executed. The user starts with a parking lot reservation request. This causes an initial instance of the `ParkingReservation` intent to be pushed on the stack which sets the current topic of discourse. Of the three members defined for the intent (place, time, and duration) the user only specifies the place ("Shedd Aquarium"). Since there is no stack element underneath, *r* remains empty. Due to the updated stack, first a confirmation prompt for the POI mentioned in the first utterance gets active, followed by a call back prompt, which asks for the time of the reservation. The modeler may have chosen to handle time requests in a dedicated sub-dialog and introduced an intent `EnterTime` for that purpose. Thus, an instance of that intent class is pushed on the stack defining a new sub-topic. Its member value is not yet set in *m*. Note that the return values *r* link the member values to those of the underlying `ReserveParking` intent. The user now chooses not to respond to the question, but to interrupt the dialog with a weather request. The system pushes the corresponding `WeatherInfo` intent with the member value "Chicago" (taken from the utterance) on the stack. The stack snapshot reflects the status of the dialog. The weather topic is currently in focus, but there is still the reservation with its time-related sub-dialog on the stack which can be returned to once the weather intent is finished.

The dialog stack model now plays the role of the information state that influences what dialog elements to activate next (cf. Figure 2). In order to control the activation of dialog elements, the property `preIntent` is evaluated again (but now in a slightly different way than before): A dialog element is considered for activation

- if the `preIntent` property is not set (has no value), or
- if the `preIntent` property value is equal to the intent *i* of the topmost stack element.

Not defining `preIntent` thus means to always activate a dialog element. Note that this condition may not be sufficient, because other properties may put further constraints on the activation of a dialog element. The effect of dialog element execution on the stack depends on the properties `postIntent`, and the binary-valued `push` and pop properties. Upon execution of a dialog element

- if the `push` property value is true, an instance of `postIntent` is put on the stack,
- the member values of the topmost stack element are updated according to the slots values of the dialog element, and
- if the `pop` property value is true, the topmost element is removed from the stack.

To achieve the dialog behavior of our example (cf. Figure 3), the modeler could add the dialog elements listed below to the talk state. Note that the specification of the recognizable text for commands and the output text for prompts is omitted in these examples. It is possible to use SRGS grammars as well as collections of example sentences for commands. Both commands and prompts can have slots that correspond to the members of the intents.

1. **command** to initiate a parking reservation request
   `preIntent = ⊥, postIntent = ReserveParking, push = true`
2. **prompt** as a confirmation for the POI to park at
   `preIntent = ReserveParking, postIntent = ⊥, push = false,`
   `member = parkPlace, confirm = true`
3. **prompt** to query the parking time
   `preIntent = ReserveParking, postIntent = EnterTime, push = true,`
   `member = parkTime, undefined = true, returnVal = {time, parkTime}`
4. **command** to initially recognize a time-related expression
   `preIntent = EnterTime, postIntent = ⊥, push = false`
5. **command** to initiate a weather request
   `preIntent = ⊥, postIntent = WeatherInfo, push = true`
6. **prompts** as an information about the resumption of the parking request
   `preIntent = ReserveParking, postIntent = ⊥, push = false,`
   `resume = true`
7. **prompt** to query the parking duration
   `preIntent = ReserveParking, postIntent = EnterDuration, push = true,`
   `member = parkDuration, undefined = true,`
   `returnVal = {duration, parkDuration}`
8. ...

In our example, commands (1) and (5) get activated at the beginning, because they don't have any specific preconditions. (1) gets recognized, and `ReserveParking` is pushed on the stack. Then the pre-conditions for the confirmation prompt (2) match and it is played, immediately followed by the call back prompt (3) for the parking time. This pushes `EnterTime` on the stack, and activates the command for time recognition (4) in addition to command (5) for the weather. Since the user issues a weather request, (5) gets recognized, and `WeatherInfo` is pushed, finally leading to the situation depicted in Figure 3.

$i = \textbf{EnterDuration}$
$m = \{(duration, \perp, false)\}$
$r = \{(duration, parkDuration)\}$

$i = \textbf{ReserveParking}$
$m = \{(parkPlace, SheddAquarium, true),$
$(parkTime, 4o'clock, false),$
$(parkDuration, \perp, false)\}$
$r = \{\}$

System: "Currently it is warm and sunny in Chicago. Let's continue with your parking reservation. At what time?"

User: "Four o'clock."

System: "How long would you like to park?"

Figure 4 – Status of the dialog stack after discourse shown right.

In addition to this explicit effect of dialog elements on the stack, an automatism ensures that the dialog does not get stuck: When no prompt can be activated, the topmost element of the dialog stack is removed, and the activation of prompts is re-evaluated. This cycle continues until an applicable prompt has been found or the stack is empty. In the latter case the dialog comes to a stop, in the former case it continues by resuming a topic which has been interrupted before. Intuitively this behavior removes a topic (intent) when there is nothing more to say about it. It is in this auto-pop stage when the return values get evaluated. To further illustrate the behavior, consider a possible continuation of the example as depicted in Figure 4. Assuming that there is some business logic to handle the weather request and retrieve the answer (not covered here), a response is generated. There is nothing more to talk about regarding the weather intent, so the element automatically gets removed from the stack, putting `EnterTime` in focus again, but it is also removed since no prompts are defined for it, going back to the `ReserveParking` intent. At that point prompt (6) is played which has a `resume` pre-condition which activates it in case the stack was popped and the topmost intent matches the `preIntent` property. Then prompt (3) is activated again leading to `EnterTime` being pushed again which in turn activates (4). This time the user says a time and (4) gets recognized. The stack is popped and the time slot is propagated to the lower level, making prompt (3) inactive. However, now prompt (7) is activated asking for the duration and pushes an instance of the `EnterDuration` intent on the stack, leading to the depicted situation.

Note that the same dialog behavior can be achieved without introducing dedicated intents for `EnterTime` and `EnterDuration`. The model could just stick to the parking intent using call back prompts in case time and duration were missing. However, by dividing a more complex dialog into dedicated sub-dialogs, the latter can be re-used as building blocks for other intents. For instance, a sub-dialog for entering the time in our example could have clarification questions in case it's unclear weather am or pm is meant. These would not have to be re-modeled for other intents that also require time entry.

3.5 Mixed initiative

Our approach can be naturally extended to the mixed initiative case when an ongoing voice interaction gets interrupted by a system-initiated dialog and is afterwards resumed. In this case pushing an intent/topic on the stack is not a consequence of a command getting recognized as it was in the examples before. Instead, some external condition (e.g. some warning signal from the navigation sub-system) occurs that gives rise to a new element being pushed on the stack (e.g. `RouteUpdate`). This is achieved by offering an interface to the dialog stack that is accessible from the outside, independently of the dialog elements. We introduced a script function for this purpose, which can be called by the modeler.[2] The modeler then needs to create appropriate dialog elements (prompts and subsequently used commands) to define the interrupting dialog.

[2]Note that EB GUIDE has a built-in scripting language the modeler has access to.

The resumption to the original interaction follows the same rules as user-initiated interruptions discussed above.

4 Prototype system

The modeling concepts described previously were implemented for our product EB GUIDE Speech Extension (SE).[3] It consists of definitions for the new properties of dialog elements used in the SE. A run-time module for maintaining the dialog stack has been added together with the business logic for stack queries and manipulations. A use case similar to the interaction examples presented in this paper has been modeled with our extension and can be demonstrated.

5 Conclusion

In this paper we suggested a dialog modeling and execution approach capable of multi-turn dialogs with mixed initiative and nested topics. The core device of the applied dialog theory is a dialog stack. The approach supports decomposing complex dialogs into smaller, re-usable parts and thus aims at reducing the modeling effort across multiple use cases. The approach was embedded in an existing dialog modeling tool which is largely based on the state graph paradigm. The solution thus combines the state-based dialog modeling for simple voice interactions with ideas more inclined to the information state update approach to dialog management for more complex interactions. Strengths of both approaches can thus be combined in one tool.

References

[1] HAREL, D.: *Statecharts: A visual formalism for complex systems*. *Science of Computer Programming*, 8(3), pp. 231–274, 1987.

[2] MASSONIE, D., C. HACKER, and T. SOWA: *Modeling graphical and speech user interfaces with widgets and spidgets*. In *Proc. of the 11th ITG Symposium on Speech Communication*. VDE, 2014.

[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] GROSZ, B. and C. SIDNER: *Attention, intentions, and the structure of discourse*. *Computational Lingustics*, 12(3), pp. 175–204, 1986.

[3]https://www.elektrobit.com/ebguide/