Developing Speech Dialogs For Multimodal HMIs Using Finite State Machines

Silke Goronzy, Raquel Mochales, Nicole Beringer

3Soft GmbH, Speech Dialog Systems Group
Frauenweiherstr. 14, 91058 Erlangen, Germany
silke.goronzy@3soft.de

Abstract

We present a tool for model-based development of multimodal interfaces. The HMI model captures all involved modalities, thus ensuring highly consistent interfaces. In this paper we focus on the development of speech dialogs. These are specified using state machines, which is in contrast to the traditional way of using flow-charts. The usage of state machines gives us the possibility to fully specify the HMI so that it contains enough information to be fully simulated without the need to connect any target applications as well as for automatic target code generation. Due to the extensive simulation capabilities usability evaluations can be conducted at very early design stages. We further explain how different dialog strategies for different user types can be developed with the help of the user modelling plug-in. The tool thus supports the whole development chain starting from design studies to specification, development and testing over usability studies and target implementation.

Index Terms: multimodal dialog tool, multiple state machines, user modelling, simulation, integrated development.

1. Introduction

Speech dialog systems evolve more and more in real world applications such as automated call centres or automotive infotainment systems. Such systems are often praised to be intuitive and natural and the users' expectations are correspondingly high. When using such systems however, users very quickly come across the systems' limitations. Especially in automotive environments where speech dialog systems are usually coupled with graphical/haptical interfaces to build multimodal interfaces some problems arise that are not easy to understand for system users. One of the biggest problems is the inconsistency between the different modalities. We have developed a tool for model-based development of multimodal human machine interfaces (HMI). All modalities are specified and developed as part of the same HMI model, thus ensuring consistency between the different modalities. The tool supports all phases necessary in (industrial) HMI development, such as design, specification, implementation of the HMI and finally generation of source code for the target platform for both graphical/haptical interface and speech dialog. In this paper we focus on the speech dialog part, i.e. how we specify, model and test speech dialogs. Many industrial dialog systems use state machines, but during the specification phase the different dialogs are still specified using flow-charts which have to be manually converted into a machine-readable dialog description for a (state-based) dialog manager. We directly specify the dialog using state-charts, thus the dialog model we create IS the complete specification. This has many advantages in terms of simulation capabilities and the possibility to incorporate a large number of test details in the speech dialog specification. The paper is structured as follows: In section 2 we describe the general architecture of our HMI modelling tool. Section 3 then gives more details on how speech dialogs are modelled. Section 4 describes the user modelling that supports the designer in defining different dialog strategies and properties for different user groups before we outline in section 5 how the tool and its simulation capabilities are used for conducting usability evaluations.

2. tresos GUIDE

tresos GUIDE is a tool for the generation of complex multimodal human machine interfaces (HMIs). It follows a model-based development approach. In contrast to other approaches, where different modalities are developed rather independently of each other, we use one HMI model to specify all modalities. For the different tasks involved in HMI development, such as designing screens, widgets, menu logic, speech dialog etc. we have specialised editors. The menu logic is defined using UML-compliant state machines. Transitions between states are triggered by events and so-called actions can be executed at transitions or when entering or leaving a state. Views are attached to states and the state-charts are illustrated with the 'live' views. All elements, such as widgets, views, speech dialog components are described by properties. These can be globally visible or private. It is possible to use multiple state machines that can be synchronised, e.g. for driving multiple displays or a display and speech dialog.

2.1. Architecture

In our architecture the HMI is completely separated from the applications, meaning that there is no direct connection between the two. The communication is realised by use of a data pool. All data that is relevant to the HMI, i.e. data that needs to be displayed or output to the user is held in the data pool. Figure 1 shows a schematic view of the tresos GUIDE architecture. Different components can access and/or modify the data in the data pool. This architecture requires that the applications have an interface to the data pool. A typical flow if the user presses a button or utters a speech command is as follows: A command or button press will usually trigger an event. If the user uttered 'CD next track', the corresponding event will cause the HMI framework to call an abstract interface to the application which will execute the command and write the relevant data for the display (e.g. current track number and elapsed time) into the data pool. The HMI is notified that values in the data pool changed and updates its view if the data is relevant (if the user switched to the navigation context in the meantime, the current CD track is not dispalyed).

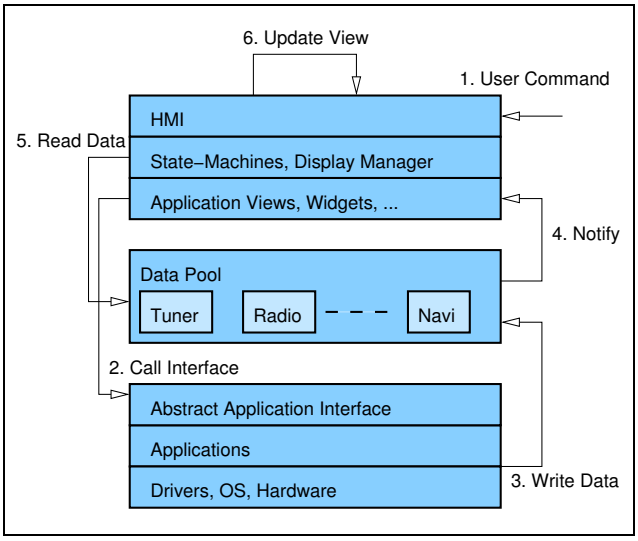

Figure 1: *Data Pool Architecture of tresos GUIDE*

2.2. HMI Simulation

Since we work with a model of the HMI, we can directly simulate it. This means that we do not need to compile the project, but changes made to a dialog can directly be simulated. This holds for all modalities, since they are all part of the HMI model. We can thus immediately get a feeling of how the HMI behaves. Usually HMI developers need to have access to the working applications before they can fully test the HMI. In our approach we can simulate all applications simply by manually writing application data to the data pool. We can thus conduct user evaluations and usability studies at very early design stages, as will be described in section 5. Traditionally such studies can only be conducted when large parts of the HMI and the system already exist.

3. Speech Dialog Specification

Just like views are attached to states in the GUI state machine (SM), we attach so-called *speech dialog (SD) components* to the states of the speech SM. An SD component holds all prompts and/or commands (specified as words or grammar rules) that are valid in this state. In most of the cases one SD component directly corresponds to one state in the speech dialog. However, it is also possible to use the same SD component at different states. A special speech dialog editor is used to specify all commands and prompts. These are treated as properties that are kept in the data pool. The dialog designer can furthermore specify whether certain 'global' commands are inherited from states further up in the state hierarchy. Doing this we can avoid that re-curring commands have to be specified over and over again. Like all properties in the HMI, also the command and prompt properties can be defined to be language-dependent. While the dialog flow is defined using concepts such as e.g. DIAL_TEL_NUM, the wording for the different languages is kept separately. All language-dependent properties can be exported for translation and imported back into the system. The system can thus immediately be simulated in different languages (if the dialog flow itself remains unchanged for the different languages).

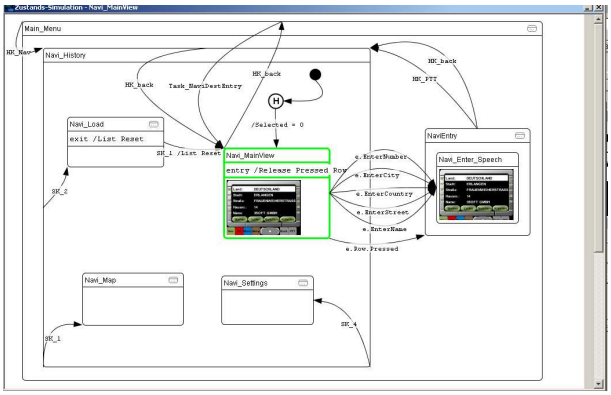

Figure 2: *State Machine for the GUI*

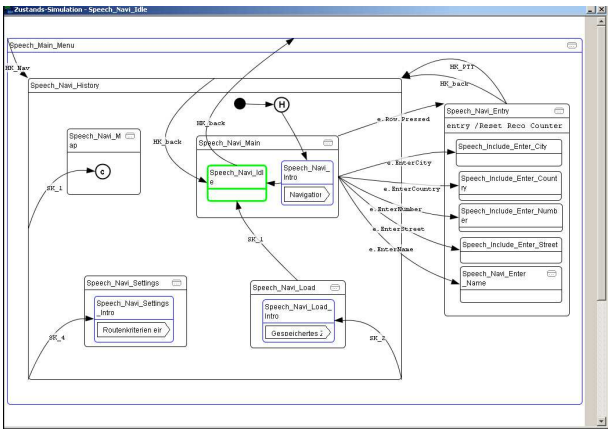

Figure 3: *State Machine for the Speech Dialog*

3.1. Speech Dialog State Machines

Similar to the GUI specification we use state machines to specify the speech dialog flow. This is in contrast to many traditional approaches of industrial systems that use finite state machines for the dialog manager but the complete specification of the speech dialogs is done using flow-charts (This specification very often needs to be manually converted into a machine-readable dialog description for the dialog manager). While a speech dialog flow can be captured quite well in flow-charts, it is usually difficult to synchronise this with other components of the HMI and to integrate all details that are needed for testing the target system. In flow-charts references to other modalities or the GUI are usually included in plain text such as 'HMI returns list of CDs inserted'. Such a textual description is difficult to automatically translate into the final overall HMI implementation or into test cases and it is difficult to automatically generate source code for the target platform. In our architecture we have direct access to e.g. the list of CDs inserted, because such information is kept in the data pool where the speech dialog can access it. We can directly model the actions of writing and reading the list of the CDs to/from the data pool and can thus also directly simulate it and generate target code accordingly. By fully specifying the behaviour of the HMI in the model, we directly have all information available that we need for testing. The system we described in [1] was using one state machine to model the GUI as well as the speech dialog. While this has the advantage that graphics and speech dialog are inherently synchronous, the drawback is that only simple dialogs can be modelled that basically allow users to speak what they see on the screen. If more

elaborate speech dialog flows are to be modelled they might have flows that are different from the GUI. The destination entry in a navigation system might serve as an example here. The GUI will be showing the same screen while in the speech dialog many different steps are modelled that allow the user to enter city, street, etc. including all kinds of help and error messages. For such kind of complex dialogs it is beneficial to model the speech dialog in a separate SM. As a consequence the tool was recently extended to support multiple SMs. Figure 2 and 3 show two state machines, the first one representing the dialog flow of the GUI, the second one representing the speech dialog flow. The states in the two SMs have different shapes. While the rectangular-shaped states in the GUI represent 'normal' states, the ones in the speech SM represent special speech dialog states that can either include one or several prompts or different vocabularies/grammars for speech recognition or both. Furthermore the GUI SM is illustrated by the 'live' screens that will be visible in the later target system while the speech SM is illustrated by the set of commands and prompts belonging to the different states. In order to synchronise the two SMs, events can be sent from one SM to the other, causing e.g. the GUI SM to change the screen. Furthermore actions can be modelled at transitions or when entering or leaving a state. An action could be to send an interface call, fire an event or set a value in the data pool. Additionally, complex conditions can be modelled.

3.2. Grammar Specification

The speech dialog designer can specify valid commands in terms of grammars for each SD component. The grammar format that we use is the ABNF version of SRGS (Speech Recognition Grammar Specification) [2] format. The designer can specify different rules and sub-rules. All rules that are specified in one SD component are compiled together at run-time to either build one grammar or separate grammar rules for each state. It is also possible to use conditions to activate certain grammar rules depending on the actual system context (such as 'navigation commands are only active in radio mode if the route guidance is running in the background'). In the future it may also be possible to use statistical language models, if more natural speech dialogs are to be specified. The designer can also use semantic tags within the grammar. Here we use SISR (Semantic Interpretation for Speech Recognition) [3]. Depending on the actual speech recogniser that is connected for the simulation the grammars are converted into the appropriate format.

3.3. Dynamic vocabularies

The designer can also specify dynamic vocabularies by referring to the data pool. He can e.g. refer to the list of radio stations that can currently be received, an information that might change frequently. Whenever the user wants to change the radio station and enters the corresponding dialog state, the list is read from the data pool and added to the current recogniser vocabulary. In the same way, dynamic information can be used for output prompts by referring to the data pool to generate e.g. something like 'Did you say *Hamburg*?' where 'Hamburg' is stored in the data pool as the most recently recognised city.

4. User Modelling

Typically in speech dialog systems there is the necessity to treat different types of users differently. For users who are very inexperienced with speech dialog systems explanations become necessary to tell them which kind of interaction is possible in the different dialog states. For experienced users of the system such lengthy explanations might soon be annoying and should be avoided. In the above case the difference in the beginner and expert user only consists in the type of output prompts and possibly also the available commands that are given or accepted in the different dialog states. However, it might also be possible that we want to provide completely different dialog flows to different users. Still, no matter if only the commands and prompts or also the dialog flow should differ, this typically has to be taken into account during each interaction. Defining different strategies at each interaction step during speech dialog design can become a quite tedious task, so the speech dialog editor offers a special plug-in for user modelling. By default three different types of users are defined: beginner, advanced and expert users. New user types can be generated. The dialog developer can then define which prompts and commands are valid for the different user types, so that e.g. very short prompts can be used for expert users and longer explanations for beginners or special states with only one user command and prompts that can only be accessed by this particular user type. Moreover, it is possible to show the dialog flows for all user types at the same time or only the one for one user; this gives a clear visual idea of the differences between one user type and the others.

4.1. User-Specific Dialog Strategies

Further adaptations for different user types can be made concerning dialog strategies. Depending on the settings, different dialog templates (i.e. 'mini-dialogs' including several states and transitions) can be applied. Figure 4 shows an example of such a template for entering a navigation dialog. Each state is described by its name and can include a prompt (first arrow-shaped box) and a set of commands (second, table-like box). In this example the dialog starts with the left state. Expert users, after the welcome message ('Hello, welcome...'), can give the command ('go to navigation') to go directly go the bottom state on the right where the destination can be entered. The upper state on the right is not used (prompt and command boxes are empty). The beginner dialog, cf. Figure 5, uses the same template. Here, after the welcome message, the user cannot give any command (command box of first state is empty), but the system goes to the upper right state first to play additional help prompts before going to the lower state on the right where the user has finally the possibility to utter 'go to navigation' to go to the destination entry. Different parameters exist for adapting the dialog strategy. For each user it can be defined whether the dialog should be more system-driven (system-initiative), or user-driven (user-initiative), or mixed (mixed-initiative). For each initiative some specific actions can be defined, e.g. what should happen after a certain user type specific timeout has passed. The system could e.g. either go back to the beginning of the dialog, wait for the user input or recommend the next action. There are more than ten possible specific actions to be defined. For example the dialog designer can also specify how the system should treat interruptions or barge-in. Barge-in can either be ignored, the system can add the new command (from the barge-in utterance) to the current one and execute first the original, then the new command or the system can directly switch to the new task. It can be furthermore defined how the confirmation strategy looks like for that type of user, i.e. whether to confirm it explicitly or implicitly or not at all. Additional parameters are how to solve understanding errors, to cancel a task, to undo the changes, etc. A user type has to be homogeneous, but there are some details that depend as much on the current task as on the user type, e.g. how long the system must

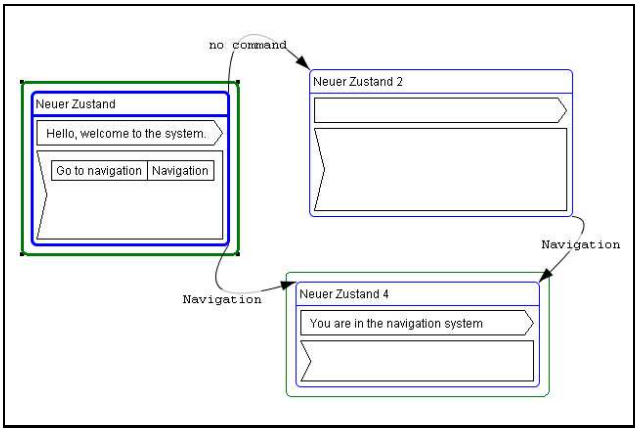

Figure 4: *Example for a template for an expert user*

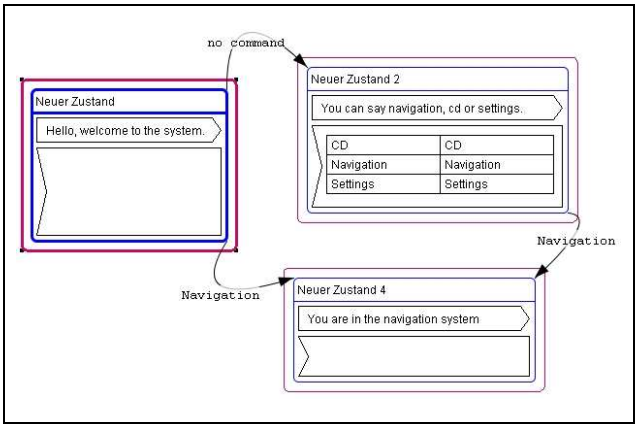

Figure 5: *Example for a template for a beginner*

wait if the user is not responding. A situation like that can either depend on the user experience or on the critical time of the task. These details will not only be related to a user but also to the current prompt or command. All user modelling module functionalities are described in detail in [4].

5. Usability Testing

As mentioned in section 2.2, the problem often arises that the system needs to be in a development state where most of the functionality is actually working properly before any kind of user evaluation can take place. However, if this evaluation reveals any problems that would require major changes to the system, this can often not be taken into account anymore because the system is too advanced already. To overcome this problem, we use the simulation capabilities of our modelling tool to conduct usability studies at very early design stages. That means the infotainment system is completely modelled and simulated for the tests. If the HMI is intended for an automotive application, one should take into account that controlling the HMI is not the primary task of the user but the secondary one. In order to simulate also this, we use the Lance Change Task (LCT) designed by Mattes [5]. The LCT is a PC-based driving simulation that measures the influence of a secondary task (in our case controlling a simulated infotainment system). The task comprises certain lane change manoeuvres that are indicated on traffic signs along the road. Between the signs the user has to keep the lane. The reaction of the users (when they are supposed to change lane) and how well they keep the lane can be measured. The results can be compared to a driving task without any secondary task and can thus give some insight into how much the secondary task distracts the drivers from their main task. Additionally, using the LCT as primary task creates a more realistic environment for testing the infotainment system, because the user cannot fully concentrate on the system but has to pay attention to the primary task, which results in e.g. hesitations and errors that would not occur if we only tested the infotainment system in isolation. The dialog flow in the HMI simulation can be recorded and evaluated later concerning errors in the dialog flow, speech recognition or the user input. This together with appropriately designed questionnaires the users have to fill after the usability test form a solid basis to evaluate HMIs in terms of usability. Of course errors in the HMI model itself (such as missing transitions, or transitions to states that no longer exist and the like) can also be detected. The detection of these kinds of errors usually doesn't require real user tests but can be performed automatically by model checkers.

6. Conclusions

In this paper we presented how speech dialogs are specified in our HMI modelling tool tresos GUIDE. The tool allows the integrated development of multimodal HMIs, thus ensuring consistency between the modalities. Although this joint development of multiple modalities is a big advantage, it is also possible to develop GUI-only or speech-only interfaces. We specify speech dialogs using finite state machines which is in contrast to the traditional approaches that use finite state-based dialog managers but flowcharts for specifying the speech dialog flow. Using the state machine concept allows us to directly simulate the multimodal HMIs without the need to connect real applications. As a consequence we can conduct usability studies at very early design phases. Our modelling tool is well suited for research as well as for production, because it allows rapid prototyping and simulation of new concepts as well as a complete specification of an industrial HMI up to automatic code generation for the target platform. The development of the tool is still being continued; future extensions of the speech dialog editor will include the support for using statistical language models and the possibility to specify frame-based dialogs. Of course also methods for mutual diambiguiation, for which approaches exist, need to be integrated into our tool.

7. References

[1] Goronzy, S. and Beringer, N., Integrated Development and On-the-Fly Simulation of Multimodal Dialogs, Interspeech 2005, Lisbon, Portugal.

[2] Speech Recognition Grammar Specification, http://www.w3.org/TR/2004/REC-speech-grammar-20040316

[3] Semantic Interpretation for Speech Recognition, http://www.w3.org/TR/2006/CR-semantic-interpretation-20060111

[4] Mochales R., Design, Specification and Implementation of Natural Dialog Strategies for a Personalised User in tresos GUIDE+SPEECH, Master Thesis, Universitat Polytechnica de Catalunya (UPC), 2006.

[5] Mattes S., The lane-change-task as a tool for driver distraction evaluation, Annual Spring Conference of the GfA/17th Annual Conference of the International-Society-for-Occupational-Ergonomics-and-Safety (ISOES), 2003