Towards Semi-Automatic Generation of Vocal Speech User Interfaces from Interaction Models

Dominik Ertl, David Raneburger
Vienna University of Technology
Institute of Computer Technology
Gusshausstrasse 27-29, A–1040 Vienna, Austria
{ertl, raneburger}@ict.tuwien.ac.at

Abstract: Manual generation of vocal speech user interfaces requires effort in different disciplines. This includes designing the interaction between a user and the system, configuring toolkits for speech input and output of the user interface, and setting up a dialogue manager for the behavior of the user interface. It requires even more effort if the user interfaces have to be created for several platforms. We present an approach to semi-automatically generate command-based vocal speech user interfaces from an interaction model. We address user interfaces for dialogue-based interactive systems. The interaction is designed once and the user interface for vocal speech input and output can be generated for different platforms. The aim is that UI designers focus on high-level dialogue design between a human and a computer instead of the low-level vocal speech UI engineering.

1 Introduction

Current research approaches for generating user interfaces (UIs) mainly address the generation of WIMP-GUIs [VdBMB+10, Van08]. Semi-automatic generation for speech-based UIs, however, is a promising approach to strive for multimodal UIs or synchronous speech UIs (speech input *and* speech output in one UI). Pure speech UIs have a niche, e.g., for visually impaired people or those that want to have their hands free while working with a UI.

This work presents an approach for semi-automatic generation of speech UIs for dialogue-based interactive systems. Our approach is based on a high-level discourse-based Communication Model [FKH+06, Pop09] and includes dedicated tool support for generating at design-time both a speech input UI and a speech output UI. As a running example, we present an interface of a mobile service robot in a shopping environment. In principle, this robot is a semi-autonomous shopping cart that supports a user while shopping [KPR+11]. This robot has a multimodal UI, however, in the context of this paper we focus on the speech input and output parts.

The remainder of this work is organized in the following manner: First, we present background information. Then we illustrate our approach for semi-automatic generation of a

erschienen im Tagungsband der INFORMATIK 2011
Lecture Notes in Informatics, Band P192
ISBN 978-3-88579-286-4

speech UI at design time. In particular, we show how a speech model for input and output is derived from a Communication Model and how this speech model is then transformed into a concrete command-based speech input and output UI. We also show how such a generated speech UI can be instantiated and used at runtime. At last, we discuss our work followed by presenting related work.

2 Background

Our approach uses a high-level *Communication Model* to describe modality-independent interaction with *Communication Models* in the sense of dialogues. Such Communication Models have *Communicative Acts* as basic units of communication. Moreover, such a model consists of Adjacency Pairs, Rhetorical Relations and Procedural Constructs which structure the interaction flow and describe alternative usage scenarios [BFK+08].

In previous work we introduced configuration files to support the transformation of the modality-independent Communication Model into a modality-dependent Communication Model [EKKF10]. In case of speech UI generation this step results in a dedicated Communication Model for speech.

A Communication Model includes also a *Domain-of-Discourse Model* that models the application domain.

Each Communicative Act has a unique identifier and specifies a so-called Propositional Content. The Propositional Content refers to action descriptions from the *Action Notification Model* [Pop09], where the actions are stubs to the core functionalities of the applications. These actions are designed and implemented by the business logic developer. Given is a Propositional Content that refers to a specific action. This means the UI designer defines that this Communicative Act triggers the referred action.

For example, the mobile shopping robot has the ability to follow a user. The Propositional Content of the *Request* Communicative Act is *followMe*, which refers to a followMe action and behavior of the robot. Another example: the robot is able to drive to a concrete product which a user has selected before in a shopping list application. Here, the Propositional Content of the Communicative Act is *goThere param::destination* (i.e. the user wants the robot to drive to this product). Here, the Propositional Content contains a parameter of type *destination*. The possible values for the parameter depend on the application logic. In the context of our shopping application the 'destination' parameters are coordinates of the concrete spatial product placements and the position of the check-out counter.

Both manually handcrafted and automatically generated UIs require code for structure and behavior [RPK+11]. In the course of UI generation, we automatically create a *modality-independent behavioral model* out of a concrete Communication Model [PFA+09] at design time. This behavior model is represented as a behavior state machine. At runtime we load a concrete version of such a behavior state machine into a Dialogue Manager component that manages the flow of interaction. In this work we present the generation of the structural representation of the speech UI, as the behavior of the speech UI is already defined through the above mentioned device-independent behavior model. We implemented

ISBN 978-3-88579-286-4

this generation based on an existing Communication Platform that is responsible for the runtime execution of the UI. This Communication Platform is a generic component and can be used for different Communication Models and their according applications.

3 Semi-Automatic Vocal Speech User Interface Generation at Design Time

In the following we describe our approach to semi-automatically generate a synchronous speech UI.

The approach consists of three major tasks that need to be performed. First, an existing high-level Communication Model needs to be transformed into a vocal speech UI. Second, the dialogue manager component needs to be configured for our Communication Platform. The third major task is the adaptation of the automatically generated artifacts by a human designer.

In Figure 1 we provide an overview of our approach. It shows the model-based transformation process for the UI generation. The modeled interaction between a human and a system is a formal and processable representation of a UI-independent interaction. Out of this Communication Model we generate a *Speech Model* which is comparable to the concrete UI (CUI) level from the CAMELEON reference framework [CCT+03]. We devised and implemented our approach based on the Eclipse Modeling Framework (EMF) [1]. So, the human designer can adapt the artifacts of the transformation steps on all levels. We believe that current approaches still require the human in the loop to create UIs with an acceptable usability since usability is a major criterion for the acceptance of a UI.

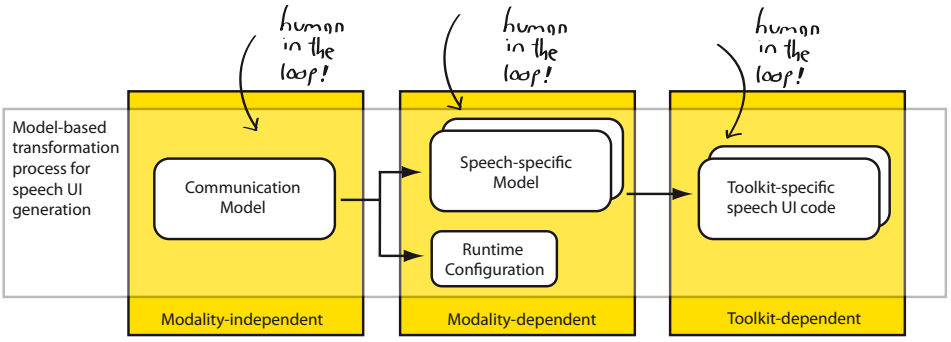

Figure 1: Overview of Semi-Automatic Speech UI Generation.

[1] http://www.eclipse.org/emf

3.1 Speech Model Generation

We use a Speech Model to describe the *structural representation of vocal speech* in a *toolkit-independent* way. The Speech Model is generated from the Communication Model and is used both for speech input and speech output. Even more, one Speech Model can be used for different toolkit dependent configurations. We first describe the Speech Metamodel and then the algorithm to generate a concrete Speech Model out of the modality-dependent Communication Model.

3.1.1 The Speech Metamodel

In our approach we defined a metamodel for the structure of each modality. In Figure 2 we present the metamodel for our speech generation approach. For graphical UIs we have another Structural UI Metamodel [FKP+09].

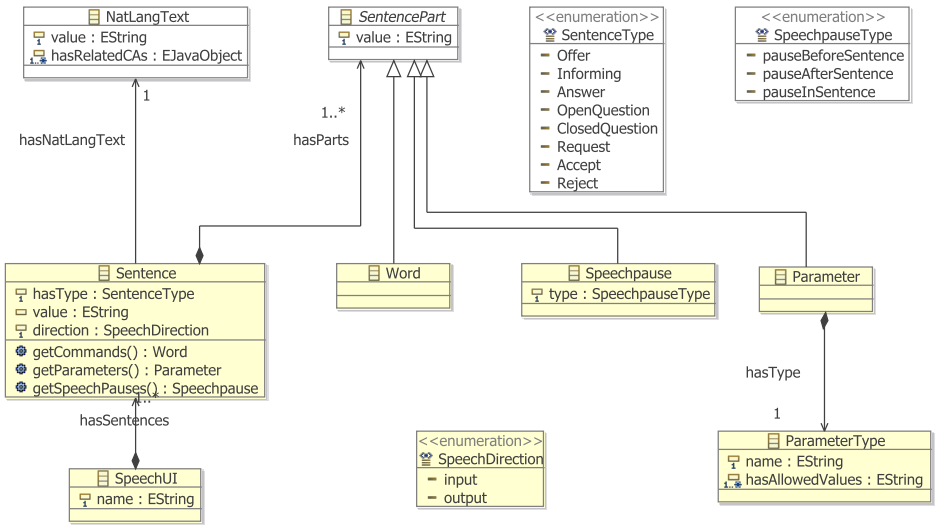

Figure 2: Speech Metamodel.

The root element is the *SpeechUI* which has several *Sentence*s. The class Sentence is of a given *SentenceType*. A Sentence refers to a natural language text, *NatLangText*. The value attribute of a NatLangText is a natural language text derived from the Propositional Content of a Communicative Act. The content of several Communicative Acts does not have to be unique within a Communication Model. Natural language texts, however, are derived from the Communicative Acts' Propositional Content. Therefore, they do not have to be unique, as the Propositional Contents are not unique either. In the Speech Model we store a unique set of natural language texts that refer to a set of Communicative Acts. At runtime we match input hypotheses from the speech input toolkit against Communicative Acts and there we consider only allowed Communicative Acts according to the current

state of the interaction (thus having a reduced set of possible Communicative Acts). The values of the enumeration SentenceType correspond to the type of Communicative Act from the Communication Model. This allows parameterizing the pronunciation of concrete speech output. For example, an Informing is formulated more neutral, whereas a Reject is more brisk. The attribute *Direction* defines if the Sentence is intended for speech input or speech output. Moreover, each sentence has 1 to n *SentencePart*s which are either of type *Word, Parameter,* or *Speechpause* plus a concrete value attribute. A Speechpause has an attribute *SpeechpauseType* that refers to the position of the pause in a sentence. Each Speechpause type might be interpreted differently in the transformation step to the concrete speech toolkit configuration and speech input or output. *Parameter* is multivalued in contrast to *Word* and Speechpause. This means, it is a variable for several (fixed) meanings. For example, in the context of our robot application a user can say *Go there left*. Here we have a concrete parameter *left* which is part of a set of direction parameters (that includes also *right*, and *front*).

3.1.2 Speech Model Generation Algorithm

The toolkit-independent Speech Model generation algorithm has four steps.

1. Take the Communication Model and make it modality-dependent by removing all Communicative Acts that do not fit for speech input or speech output. Then create a list of all Communicative Acts uttered by the user, discarding the ones uttered by the system.

2. Take the natural language text of the Communicative Acts, assuming that the natural language texts fit well as a first assumption for the creation of the Sentences in the Speech Model.

3. For each natural language text create a Sentence. The natural language text is split according to symbolic delimiters. Each split string is a SentencePart in the Sentence. A SentencePart is either a Word, a Speechpause or a Parameter.

4. Consider the possible values of param for speech input when a *param::[object]* as a parameter element is a SentencePart of a Sentence; the user is *not* intended to say, e.g., *select param::destination*. Instead, she should say *select apple* or *select banana*. So, add the names of the destinations to the list of words from the more complex content above. In the model, parameter instances are a specialization of SentencePart.

The resulting Speech Model serves now as a basis for further transformation into a concrete speech input or output configuration for platform-dependent speech input and output toolkits.

ISBN 978-3-88579-286-4

Type of SentencePart	Value of SentencePart
Speechpause	pauseBeforeSentence
Word	meet
Word	me
Word	at
Speechpause	pauseInSentence, value: 250 (if the sentence is rendered for speech output, the synthesizer waits 250 milliseconds before the following SentencePart is output)
Parameter	param::destination (the parameter can be of any value that is stored in the list of destinations in the Domain-of-Discourse Model)
Speechpause	pauseAfterSentence

Table 1: Type of SentenceParts and Value of SentenceParts for Example Propositional Content.

3.1.3 Deriving Natural Language Text out of the Propositional Content

For our speech generation we need natural language texts that are derived from the Propositional Content. We use these natural language texts in the later described Speech Model. The architecture style for generating such natural language text is basically a pipes-and-filter one, where the Propositional Content is parsed, split and reformatted. Considering the two examples above, the Propositional Content *followMe* results in two morphems *Follow* and *Me*; and *goThere param::destination* results in three morphems, namely *Go*, *There* and *param::destination*. Even if the natural language text could look similar to the Propositional Content, we need an explicit representation. It cannot be guaranteed that actions from the Action Model have meaningful names. For example, a Propositional Content refers to an action named *ab112*. Here, our mechanism does not derive an understandable natural language text, but leaves it to the UI designer to name it properly and exchange the placeholder string *ab112* of the natural language text with another one (like 'stop robot', if *ab112* is the stop action).

Here we present an example for a complete sentence that is part of a concrete Speech Model. This sentence is toolkit-independent and rendered later into concrete toolkit-dependent representation (comparable to final source code of a WIMP-GUI). Consider a Communicative Act of type Request with the Propositional Content *meetMe param::destination*. First, we parse the Propositional Content into a NatLangText *meet me - param::destination*. A UI designer adapts this NatLangText and makes it more meaningful for a human user, leading to a NatLangText with value *meet me at param::destination*. Let us explain how a concrete representation of this Sentence looks like according to our Speech Metamodel. The Communicative Act can be uttered by the user, so the attribute direction is set to input. The SentenceType here is of type *Request* and has an ordered list of SentenceParts which we present in Table 1.

ISBN 978-3-88579-286-4

A complete SpeechUI consists of several such sentences. All sentences together form the structural representation of the speech UI.

3.2 Concrete Speech User Interface Generation

The modality-dependent but toolkit-independent Speech Model is the basis for the final transformation into a concrete vocal Speech UI for input and output. To get such a concrete UI, we need dedicated toolkits both for speech input recognition as well as for speech out synthetization. Such speech toolkits are comparable to the Qt- or Java Swing library for GUI rendering.

3.2.1 Generation of Vocal Speech Input

In our approach we use the grammar-based Julius[2] speech recognition toolkit because it is Open Source and widely distributed. The Julius speech toolkit is a multi-language speech processing tool for command-based input. Julius is not intended for natural language processing. The result of the generation process are concrete configuration files for Julius.

The complete generation process to the Julius configuration from a Communication Model is depicted in Figure 3.

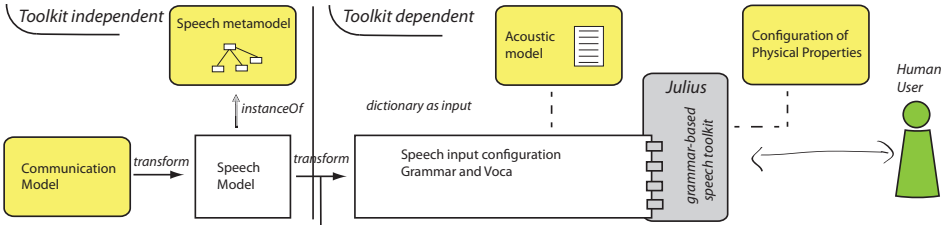

Figure 3: Automatic Speech Generation Based on the High-Level Model.

On the left the figure shows the generation of the toolkit-independent Speech Model from the Communication Model. The following transformation step creates the toolkit-dependent configuration files.

Types of configuration files: Julius needs three configuration files to get a concrete instance of a Julius-based speech interface running. We generate two types of them from the Propositional Content. The third file configures the physical properties of the Julius toolkit and is not automatically generated but a template filled with basic data from the Communication Model. More concrete, these files are:

- *Grammar File*: A grammar is a set of rules that describes a sequence of tokens that

[2] http://julius.sourceforge.jp/en_index.php?q=en/index.html

are expected at a given input. This file contains a finite-state symbolic representation of the speech input.

- *Voca File*: This file contains the pronunciation — the phoneme sequence. Phonemes are the smallest segmental unit of sound employed to form meaningful contrasts between utterances[3]) for all words within each category defined in the grammar.
- *Physical Properties Julius Configuration*: Additionally, we generate a template *[commmodelname].jconf* file that is the overall runtime configuration of Julius. This template is loadable without any further required manual adaptations. This file contains a lot of configurations, like file paths, decoding options, the input type (microphone or line-in), etc (mainly physical properties). Here, the UI designer is required to adapt it according to the needs of the system.

When the grammar and voca files are created we trigger the execution of an external script from the Julius toolkit. This script generates the finite state automate and the coupling of the grammar/voca files for the speech recognizer required for runtime.

Using the Acoustic Model: Additionally, we need an acoustic model for a given language (in our example this is English) that has a predefined set of triphones (sequences of phonemes). Such triphones are used to form words that have a meaning in the chosen language. We generate the grammar and voca files out of the speech model instance based on the Xpand[4] model2text transformation tool. For each Sentence in the SpeechModel we search for each SentencePart if the word is contained in a dictionary that is part of the acoustic model. The dictionary contains a mapping between recognizable words and their triphone representation. For example, the Sentence *edit shopping list* is represented with the following triphones *edit: eh dx ax t, shopping: sh aa p ix ng, list: l ih s t*. So, the dictionary contains all words that are recognizable in principle by the Julius toolkit. We use an inferer module to allow for some degree of blurring between the SentencePart and the dictionary elements; e.g., if the dictionary does not contain *pineapple* but *apple*, *apple* is chosen as alternative. If a representation of the SentencePart is not found in the dictionary, we use the SentencePart that was not found as a placeholder and mark it in the grammar/voca files. As long as there is no word-triphone matching in the dictionary, Julius cannot recognize the word at runtime. However, the UI designer can manually add a suitable set of triphones to the dictionary. Alternatively she can replace the SentencePart in the Speech Model using an already existing word in the dictionary. Moreover, we add to each generated grammar a set of typical and often used triphones set (*hello*, *yes*, *no*, etc.) to reduce the false positives rate. If we use a reduced grammar, artificially adding false positives enhances the recognition of the desired words. Even a *hello* could in a noisy environment be recognized by Julius as *banana*, so it is common practice to automatically add a set of typical false positives.

[3] `http://en.wikipedia.org/wiki/Phoneme`
[4] `http://www.eclipse.org/Xpand`

3.2.2 Generation of Vocal Speech Output

The Speech Model is also the basis for our concrete speech output representation. The generation of the configuration consists of two steps:

1. Retrieve the list that contains all Sequences in the Speech Model that are of direction *output*. For each of these Sentences take the ordered list of Words, Parameters, and Speechpauses and couple them to an output element.
2. Retrieve the list of related Communicative Acts of the Sentences' natural language representations. Store a mapping of Communicative Acts and natural language text in a dedicated configuration file *[comm-modelname].tts*.

An example entry in this file is *17 - ⟩ arrived at ⟨ SILENCE MSEC="250" ⟩ name*. Here *17* is the ID of the Communicative Act, *arrived* and *at* are Words, ⟨ *SILENCE MSEC="250"* ⟩ the pauseInSentence and *name* a parameter field that has to be filled in context-dependent at runtime from the application logic.

This file is loaded at startup from the fission module [EFK10] of the Communication Platform and related to a concrete speech output synthesizer. In the course of our work we have used it for the toolkits Festival[5] and FreeTTS [6].

4 A Generated Vocal Speech User Interface At Runtime

The speech UI is part of a Java-based *Communication Platform* [PFA+09] for multimodal UIs. This platform contains, among components for fusion and fission in case of a multimodal UI, a runtime component of a Dialogue Manager that interprets the state of interaction according to the Communication Model. Each toolkit (like Julius) is integrated into the platform via a stub that implements a modality-independent interface of the platform to normalize and structure the information flow between the modality toolkit and the Communication Platform. The common message units within the platform are Communicative Acts. The Communication Platform is coupled to the underlying application logic through the Communication Model.

In Figure 4 we depict the architecture of our synchronous speech UI at runtime. The toolkits for speech input and speech output both require the implementation of a stub that couples the toolkits with the platform. Here, the 'SR stub' is the speech recognizer stub and the 'SS stub' is the speech synthetization stub.

For example, when the shopping robots' application is started, we have the following (properly configured) components running: The application logic (distributed on several software layers, from motor control up to the management of a shopping list), the Communication Platform (including the Dialogue Manager), and the toolkits for speech input (Julius) and speech output (Festival). In principle, the speech input and speech output

[5] http://www.cstr.ed.ac.uk/projects/festival
[6] http://freetts.sourceforge.net

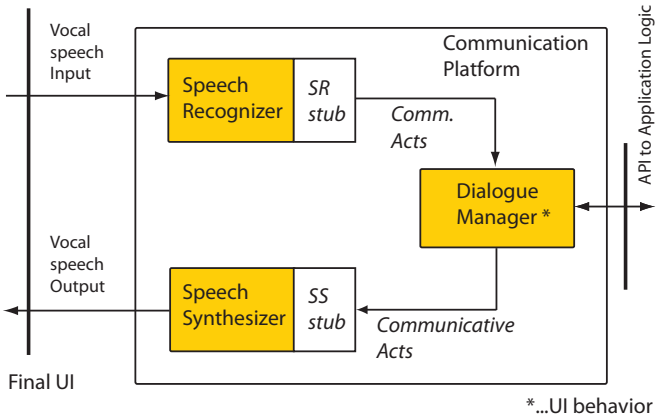

Figure 4: Architecture of User Interface Runtime Platform.

toolkits can be replaced with other toolkits that have the same modality but provide a better recognition rate (which is crucial for speech input). It is important to mention, that the clearly defined interfaces allow *decoupling* of concrete modalities and Communication Platform on the one hand, and several kind of applications and the Communication Platform on the other hand. This approach makes our approach of speech UI generation attractive for several underlying hardware platforms (e.g., robots, Smartphones, PCs, etc.) as well as different types of applications.

5 Discussion

Our method takes as input a formally designed interaction model which can be transformed into several modalities, like speech input and output. Moreover, the toolkit-independent Speech Model can be used for several speech input and output toolkits. According to the chosen (hardware) platform and available toolkits one just has to implement the model2text-part of the generation process for this toolkit. For example, if a UI designer wants PocketSphinx instead of Julius, she writes the Xpand-code to create the structure and content of the configuration files of PocketSphinx with the SpeechModel as a basis. The more platforms are supported the lower is the expected effort for writing the model2text part. Another reason for choosing a different toolkit might be the recognition rate or speech output quality of a toolkit. As our approach does not influence both aspects of a speech UI, a designer is free to select the one that fits her needs best.

An advantage of our approach is that the representation of the behavior (behavior model) of the speech UI has to be derived only once for all modalities and platforms. This behavior model is loaded at runtime in the Dialogue Manager which manages the flow of interaction according to user input and machine output. Moreover, this behavior model is modality-independent, so it is used for other modalities like WIMP-GUI [RPK+11] or gestures as

well.

We support the human in the loop on several steps of our generation process. The main points here are the inclusion of a designer to create an efficient UI. Every application depends on the context of use, if it is for trained users or a public area for the men/women on the street, the size of the grammar, the ability of the speaker etc. Other ways known from HCI for user studies and performing experiments are helpful to gain quantitative and qualitative results to iteratively improve the interaction and UI.

In our approach we only address command-based speech UIs. A limitation of our semi-automatic generation approach is that it heavily depends on useful verbs and nouns used to describe the actions referred in the Propositional Content. We use the action descriptions as symbols. If these descriptions are not precise enough the UI designer is forced to heavily adapt the Speech Model and/or concrete configurations. Obviously, this would outperform the benefits from automatic transformation in contrast to manually create a UI. To achieve acceptable results, the Action Notification Model requires some 'formalized language' where the naming is not arbitrary and ambiguous but related to the natural language.

Another shortcoming in the current approach is the implementation of the mechanism to match words in the Speech Model with the ones in a toolkits' dictionary. On a technical level this is now a comparison of strings with some blurring. Ideally, we would use here ontologies and semantic technologies for the dictionaries and Speech Model. This allows comparing taxonomies and leads to an expected better matching. The benefits are then manifold. First, the naming of actions has more degrees of freedom, requiring a less formal language (which is also helpful if the action designer and UI designer are from different 'working cultures'). Second, it allows easier internationalization of a speech UI (different language tags for one symbol). Finally, due to the taxonomy there are more options to choose from for a distinct symbol for the UI designer (like *fruit*, *apple*, or *KronprinzRudolf*) in the Speech Model and dictionary.

6 Related Work

Speech UIs have been studied for decades. As we address synchronous speech UIs, we build on the results of previous work for speech input (recognition) and speech output (synthetization). For example, Shneiderman [Shn00] discusses the limits of speech recognition and points out that designers must understand acoustic memory and prosody. This is important for our work as we focus on command-based speech input, in contrast to natural language recognition. Configuring speech output also requires some expertise: the work of Tomko et al. [STR05] points out the importance of speech output design to ensure that the system is able to convey information and express concepts to the user clearly and accurately. To ensure this issue properly, we consider a human designer for refining the concrete output description. A dialogue-based UI typically allows some turn-taking, mixed-initiative, etc. which a a dialogue manager has to handle properly. In contrast to the work of Fuegen et al. [FHW04], who aim for a tight coupling of a speech recognizer with a dialogue manager, we designed a modality-independent dialogue manager. This dialogue

ISBN 978-3-88579-286-4

manager implements the behavior of the interaction system, through runtime interpretation of the Communication Model. This dialogue manager allows for different input and output toolkits on different platforms.

Previous work of automatic speech UI generation typically addresses multimodal Web UIs [SVM08, PG06]. They commend the use of VoiceXML for the resulting UI description, as VoiceXML can be used alone for a speech UI or together with rendered HTML for a multimodal UI. In our work, however, we also aim for generated speech UIs for non-Web applications, like the synchronous speech UI of a robot, thus we extend the state-of-the-art. As high-level model these approaches use task models as CTT. Communication Models are modality- and device-independent (physical device), whereas task-models are modality-independent but device-dependent. Another approach is the work of Plomp et al. [PMI02] that uses UIML to define a generic dedicated widget vocabulary to generate GUI and speech from a single UI description. Their approach can be used to generate several voice-based formats, most notably VoiceXML. We also define a speech model, however, additionally we show how we devised and implemented a generation process.

7 Conclusion

In this paper we present an approach to semi-automatically generate a synchronous speech UI for dialogue-based interactive systems. To our best knowledge, there is no other approach available where a command-based speech UI can be created semi-automatically from a modality-independent interaction model. This approach shall encourage UI designers to focus on high-level dialogue design between a human and a computer instead of low-level speech UI engineering. So, the UI designer is free to concentrate on an essential part of HCI, the dialogue itself. Moreover, we expect that our approach makes it affordable to add other toolkits for speech input and speech output if, e.g., the recognition rate of one toolkit is not convincing or another hardware platform requires a different toolkit.

Acknowledgements

Special thanks are due to Hermann Kaindl and Edin Arnautovic for comments on earlier drafts of this paper.

References

[BFK+08] Cristian Bogdan, Jürgen Falb, Hermann Kaindl, Sevan Kavaldjian, Roman Popp, Helmut Horacek, Edin Arnautovic, and Alexander Szep. Generating an Abstract User Interface from a Discourse Model Inspired by Human Communication. In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS-41)*, Piscataway, NJ, USA, January 2008. IEEE Computer Society Press.

[CCT+03] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A Unifying Reference Framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289 – 308, 2003. Computer-Aided Design of User Interface.

[EFK10] Dominik Ertl, Jürgen Falb, and Hermann Kaindl. Semi-automatically Configured Fission for Multimodal User Interfaces. In *ACHI*, pages 85–90, 2010.

[EKKF10] Dominik Ertl, Sevan Kavaldjian, Hermann Kaindl, and Jüergen Falb. Semi-Automatically Generated High-Level Fusion for Multimodal User Interfaces. In *Proceedings of the 43nd Annual Hawaii International Conference on System Sciences (HICSS-43)*, Piscataway, NJ, USA, 2010. IEEE Computer Society Press.

[FHW04] Christian Fügen, Hartwig Holzapfel, and Alex Waibel. Tight coupling of speech recognition and dialog management - dialog-context dependent grammar weighting for speech recognition. In *Proceedings of 8th International Conference on Spoken Language Processing (INTERSPEECH-2004, ICSLP)*, pages 169–172, 2004.

[FKH+06] Jürgen Falb, Hermann Kaindl, Helmut Horacek, Cristian Bogdan, Roman Popp, and Edin Arnautovic. A discourse model for interaction design based on theories of human communication. In *Extended Abstracts on Human Factors in Computing Systems (CHI '06)*, pages 754–759. ACM Press: New York, NY, 2006.

[FKP+09] Jürgen Falb, Sevan Kavaldjian, Roman Popp, David Raneburger, Edin Arnautovic, and Hermann Kaindl. Fully automatic user interface generation from discourse models. In *Proceedings of the 13th International Conference on Intelligent User Interfaces (IUI '09)*, pages 475–476. ACM Press: New York, NY, 2009.

[KPR+11] Hermann Kaindl, Roman Popp, David Raneburger, Dominik Ertl, Jurgen Falb, Alexander Szep, and Cristian Bogdan. Robot-Supported Cooperative Work: A Shared-Shopping Scenario. *Hawaii International Conference on System Sciences*, 0:1–10, 2011.

[PFA+09] Roman Popp, Jürgen Falb, Edin Arnautovic, Hermann Kaindl, Sevan Kavaldjian, Dominik Ertl, Helmut Horacek, and Cristian Bogdan. Automatic Generation of the Behavior of a User Interface from a High-level Discourse Model. In *Proceedings of the 42nd Annual Hawaii International Conference on System Sciences (HICSS-42)*, Piscataway, NJ, USA, 2009. IEEE Computer Society Press.

[PG06] Fabio Paternò and Federico Giammarino. Authoring interfaces with combined use of graphics and voice for both stationary and mobile devices. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI 2006)*, pages 329–335, New York, NY, USA, 2006. ACM.

[PMI02] C.J. Plomp and O. Mayora-Ibarra. A Generic Widget Vocabulary for the Generation of Graphical and Speech-Driven User Interfaces. *International Journal of Speech Technology*, 5:39–47, 2002. 10.1023/A:1013678514806.

[Pop09] Roman Popp. Defining communication in SOA based on discourse models. In *Proceeding of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*, pages 829–830. ACM Press: New York, NY, 2009.

[RPK+11] David Raneburger, Roman Popp, Hermann Kaindl, Jürgen Falb, and Dominik Ertl. Automated Generation of Device-Specific WIMP UIs: Weaving of Structural and Behavioral Models. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '11)*, New York, NY, USA, to appear June 2011. ACM.

ISBN 978-3-88579-286-4

[Shn00] Ben Shneiderman. The limits of speech recognition. *Communications of the ACM*, 43(9):63–65, 2000.

[STR05] A. Toth J. Sanders A. Rudnicky S. Tomko, T. K. Harris and R. Rosenfeld. Towards efficient human machine speech communication: The speech graffiti project. *ACM Transactions Speech Language Processing*, 2, 2005.

[SVM08] Adrian Stanciulescu, Jean Vanderdonckt, and Benoit M. Macq. Paving the Way of Designers - Towards the Development of Multimodal Web User Interfaces. *ERCIM News*, 2008(72), 2008.

[Van08] Jean M. Vanderdonckt. Model-Driven Engineering of User Interfaces: Promises, Successes, and Failures. In *Proceedings of 5th Annual Romanian Conf. on Human-Computer Interaction*, pages 1–10. Matrix ROM, Bucaresti, Sept. 2008.

[VdBMB+10] Jan Van den Bergh, Gerrit Meixner, Kai Breiner, Andreas Pleuss, Stefan Sauer, and Heinrich Hussmann. Model-driven development of advanced user interfaces. In *Proceedings of the 28th of the international conference extended abstracts on Human factors in computing systems*, CHI EA '10, pages 4429–4432, New York, NY, USA, 2010. ACM.

ISBN 978-3-88579-286-4