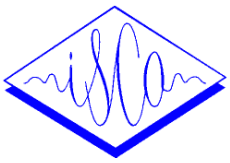
ISCA Archive
http://www.isca-speech.org/archive



ITRW on Error Handling in Spoken Dialogue Systems
Château d'Oex, Vaud, Switzerland
August 28-31, 2003

Handling errors and determining confirmation strategies – an object-based approach

Michael McTear[1], *Ian O'Neill*[2], *Philip Hanna*[2], *Xingkun Liu*[2]

[1]School of Computing and Mathematics
University of Ulster, Northern Ireland
mf.mctear@ulster.ac.uk
[2]School of Computer Science
Queens University Belfast, Northern Ireland
{i.oneill,p.hanna,xingkun.liu}@qub.ac.uk

Abstract

Errors can occur at every level of a dialogue, from the recognition of what words were spoken to the understanding of the intentions behind the words. Our approach to error-handling assumes that errors cannot be avoided in spoken dialogue and that it is more useful to focus on methods for detecting and dealing with miscommunication when it occurs. An object-based architecture is presented that supports decisions as to what sorts of confirmations should be used at various stages within a dialogue and how a dialogue agent can address miscommunication arising from user misconceptions.

1. Introduction

Errors can occur at every level of a dialogue, from the recognition of what words were spoken to the understanding of the intentions behind the words. To date most effort has been directed towards the detection and correction of speech recognition errors. However, as speech recognition accuracy improves, errors at other levels of the dialogue process will require attention, particularly in the case of advanced dialogue systems that support more flexible interaction than most current system-directed approaches.

Our approach to error handling can be characterized as an agent-based approach. We begin with the assumption that errors are a natural occurrence in spoken communication, both in human-machine dialogues as well as in human-human communication. For this reason the solution cannot be just a matter of preventing or minimizing errors through careful design, as would be the case in a GUI-based system. Instead what is required is an approach that treats errors, at whatever level, as unavoidable and that has methods for detecting and dealing with miscommunication when it occurs. Our approach is based broadly on the theory of grounding, which states that participants in a conversation collaborate to establish and maintain common ground and thus seek to avoid miscommunication and to deal with it appropriately when it occurs [1, 2]. In this view error handling involves deciding what to do when an error is detected (or suspected). In order to make such decisions, the system makes use of all the information it has available to it and assesses the costs and benefits of repairing the miscommunication. This information, often referred to as the system's *information state*, may include information about what has been said in the dialogue so far, current agendas and priorities, recognition confidence levels, and various other sources of information that together contribute to the agent's dialogue strategy.

This paper is structured as follows. The next section introduces the issues of error detection and error correction in spoken dialogue. Following this we introduce an object-based architecture and the representations that are used to handle transactions with a user. We then show how our system uses these representations to determine what sorts of confirmations should be employed at various stages within a dialogue. In the final section we examine some ideas for future work.

2. Errors in spoken dialogue

Most work to date has addressed speech recognition errors that can be explained in terms of a number of factors, such as signal quality, confusable words, unusual accents, or disfluencies in spontaneous spoken language. However, as speech recognition accuracy improves, errors at other levels – syntactic, semantic, discourse, and pragmatic – will become more apparent. Moreover, it may not necessarily be possible to determine in each case which level is involved. Consider the following example.

System: *Where do you want to travel to?*
User: *To London on Friday*
System recognizes: **To London around five** (1)

This appears to be a simple case of misrecognition, in which the user has said 'on Friday' and the system has recognized 'around five'. However, the effects of this misrecognition can be propagated to other levels, particularly if the error is not detected at the outset, for example, by using an explicit confirmation. Given that the error is not detected and corrected in this way, the following situation will arise:

Table 1: System and user beliefs

System's beliefs	bel(S,said(U, 'around five')) bel(S,bel(U,said(U, 'around five')))
User's beliefs	bel(U,said(U, 'on Friday')) bel(U,bel(S,said(U, 'on Friday')))
System's model	bel(S,time:5,day:unknown) bel(S,bel(U,time:5,day:unknown))
User's model	bel(U,time:unknown,day:Friday) bel(U,bel(S,time:unknown,day:Friday))

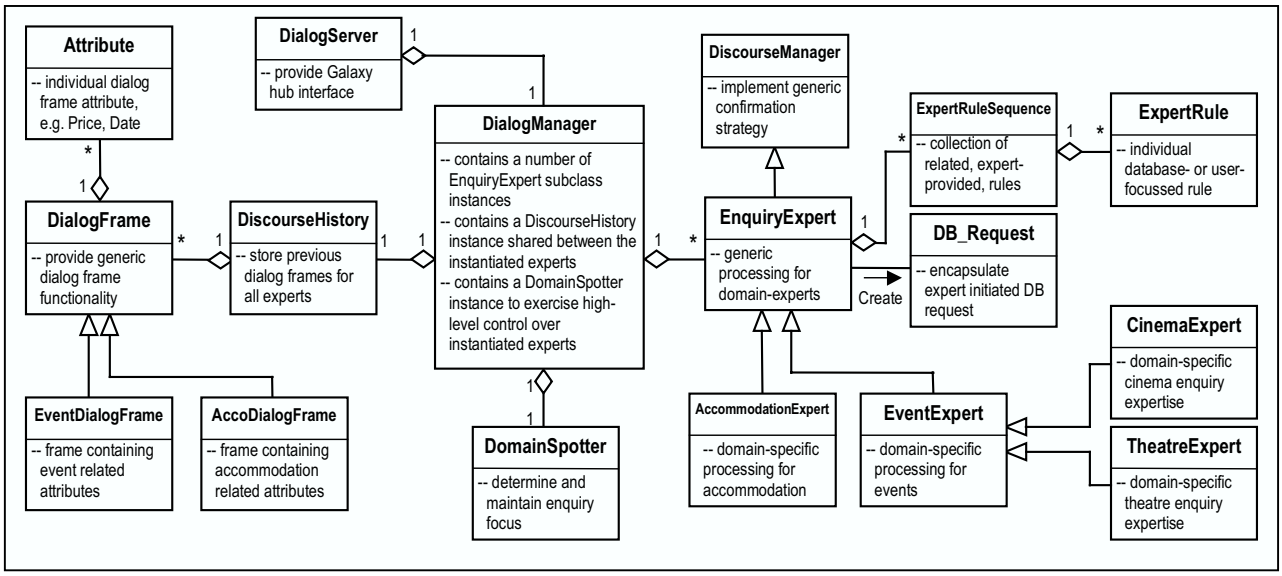


Figure 1: High level UML class diagram for the dialogue manager

As shown in Table 1, as a result of the misrecognition the system and user will have different sets of beliefs and different models of the status of the dialogue. These may persist for some time and indeed may never be resolved. Confusion will arise if on the basis of its model the System says something like 'there are no flights at that time' or 'at what time do you want to leave' as these questions will not make sense to the User given the User's beliefs and model.

Consider now a second example:

User: *I'd like a flight to Geneva arriving at 9.*
System: *There's no flight arriving at that time.*
There is a flight that arrives at 9.45. (2)

What sort of error has occurred here? On the one hand, the user may have asked for a flight that does not exist, in which case the system has co-operatively found an alternative that might satisfy the user's requirements. However, the system could also have misrecognised the value '9' in the user's utterance (if, say, the user had actually said '1'). In that case the system would have returned a flight that was probably not going to satisfy the user's requirements and indeed might have caused considerable confusion or annoyance. In other words, we cannot be sure if a miscommunication has occurred, and if we think there is a miscommunication, we cannot be sure what type of error it was. This being the case, handling miscommunication needs to be seen as a process of decision making under uncertainty. The following sections introduce an approach to handling miscommunication in terms of an object-based architecture with representations that handle the sorts of uncertainties that occur in spoken dialogue together with mechanisms for resolving the miscommunication.

3. An object-based approach

Our approach builds on the DARPA Communicator architecture, and specifically on the components supplied with the CU Communicator [3]. We have removed the dialogue management components from this system and replaced them with Java components of our own. Figure 1 gives an overview of the main components of the dialogue management system. As this system has been described in detail elsewhere [4], we will focus only on those elements that are relevant to the present paper. Our initial Java implementation concentrates on handling accommodation and events. An earlier Prolog++ prototype dialogue manager [5,6] dealt with events and travel. Our aim is to extend the Java implementation to encompass all these and other domains.

3.1. DiscourseManager

The DiscourseManager determines the system's generic dialogue behaviour in response to the user's utterances. Typically new information from the user is confirmed implicitly. However, other factors may affect this strategy, such as the status of the information supplied by the user. In order to assess how to proceed, the DiscourseManager makes use of a DiscourseHistory that maintains a record of the evolving dialogue in terms of a stack of DialogFrames, each of which comprises a set of Attribute objects relevant to the transaction.

3.2. DiscourseHistory, DialogFrame, Attribute

The DiscourseHistory contains methods that assist the DiscourseManager in adding DialogFrames to and retrieving DialogFrames from the stack. The DialogFrame consists of a set of attributes relevant to the transaction, which might, for example, be an event or an accommodation booking. Each Attribute object comprises a number of data values – attributeName, attributeValue, confirmationStatus, discoursePeg, and systemIntention – that inform the system of the information that is required to complete the transaction and of the status of this information. It is on the basis of this information that the system can decide whether to ask for more values, and whether and how to confirm values that have been elicited. For example, an attribute in a dialogue frame for flights might take the following form:

attrib(from, belfast, new_for_system, 0, confirm) (3)

The first and second arguments indicate that the information is about a *source* and a *location*, while the third, fourth, and fifth arguments refer to the *confirmation status* of the attribute value, its *discourse peg* and the *system intention* concerning it.

3.3. Confirmation status, discourse peg, system intention

The *confirmation status* of attributes is a measure of the status of the attributes within the dialogue [7]. The following list of statuses is used in the current Java-based system:

new for system, *inferred by system*, *repeated by user, modified by user*, *negated by user, modified by system, negated by system* (4)

Alongside the processing of the attributes' *confirmation status*es, the 'discourse peg' of an attribute is incremented by 1 when the user repeats a value, zeroed if the value is modified, set to -1 if the value is negated. The aim here is to ensure that every attribute has been adequately confirmed (in the Java prototype its peg must simply be set to a value greater than zero) before it is used to further a transaction. Completion of the transaction requires that key information be reconfirmed – so that the discourse pegs for key pieces of information will be set greater than 1.

Every attribute uttered by the user must be repeated once or explicitly confirmed by the user to be considered confirmed by the system. Attributes that are negated or changed by the user are queried, before they are considered adequately confirmed. Only confirmed attribute values are considered for use with the system's request templates and associated rules.

The confirmation status and discourse peg are used to determine the system's intentions. The principal intentions are *confirm* (for new values), *repair confirm* (for a modified value), *repair request* (for a negated value) and *specify* (to have the user specify a required value). A new state is then pushed on to the discourse stack, and in its next dialogue turn the system will generate an utterance corresponding to the intention.

4. Deciding whether and how to confirm

The confirmation status of an attribute together with the discourse pegs enable the system to decide on its next actions in the dialogue based on what the user has said and on the status of this information within the dialogue. Once a particular level of confirmation has been reached (a 'confirmedness threshold'), the system can decide how to use information supplied by the user – for example, to complete or to further the transaction. The following is an example of the sort of algorithm that is used to complete a transaction given a set of confirmed and reconfirmed pieces of information. For example, if the system has reconfirmed the place of departure, the destination, the day of departure and the departure time, and if a final check with the instance's database indicates that the combination of data is valid, then the system can proceed with issuing a ticket. In a more structured form the *template check* representing that process runs as follows:

```
IF
  (the discourse pegs for
   departure point, destination,
   day and departure time are > 1
  AND
   the schedule includes a service for
   departure point, destination,
   day and departure time)
THEN
   generate a system
   utterance confirming a
   reservation for
   departure point, destination,
   day and departure time.
                                    (5)
```

Alternatively, if the system has all the required information except, say, a departure time, the *template check* may indicate that prompting the user for the departure time would be the next appropriate step.

The transaction rules that have been described so far are *user-focussed rules* that are used to trigger the system's response to specific combinations of information supplied by the user and to determine whether and how to confirm them based on their confirmation status. Currently the information to be elicited from the user is determined by the slots to be filled with a particular DialogFrame. In the future we plan to investigate how our approach can be generalized to accommodate more open-ended mixed-initiative dialogues.

5. Addressing user misconceptions

In example (2), presented above, we introduced the situation where the system is unable to retrieve information from the database and attempts to offer viable alternatives. The problem may have arisen because the values supplied by the user are invalid, due possibly to a user misconception. Alternatively, the system may have misrecognised the values that the user supplied. We refer to the rules involved in resolving thus type of miscommunication as *database-focussed rules*. Our current approach does not attempt to distinguish between the different possible sources of the problem, but rather simply attempts to relax constraints in the query as understood by the system in order to get some sort of match that might satisfy the user's requirements. A simple example of a relaxed query for a class of hotel in a particular location specified by the user would be as follows:

IF (failed search was to find *accommodation name*
[e.g. Hilton, Holiday Inn, etc.]
AND constraints were *location Belfast* and *class four-star* and *accommodation type hotel*)
THEN relax constraint *class four-star* and re-do search (6)

The user can then accept or reject the system's suggestion. However, if the miscommunication was due to a misrecognition by the system, then the system's suggestion may appear confusing to the user. We discuss some ways in which this issue may be addressed in the final section.

6. Future work

The representations that we have described can be compared with those used in Information State theory [8]. The DialogFrame encodes information about the current state of the dialogue, such as what the user said and the confirmation status of the values that have been elicited. The object-

oriented architecture enables new domains to be added to the system as expert subclasses.

There are two new directions that we would like to investigate. Firstly, we would like to explore the relationships between the different items of information encoded in the DialogFrame. Currently our approach is relatively simple, in that we use discourse pegs and confirmation statuses to keep track of the confirmedness of attributes. However, there is much more information available to the dialogue manager, such as the confidence scores of the utterances recognized by the system and the relevance of those utterances to the current dialogue. We need some way of combining these different sources of information to enable the system to determine its strategies based on all the information it has currently available as well as some estimate of the usefulness and reliability of each item of information. The most developed approach to this issue that we are aware of is the Conversational Architectures project in which a hierarchy of Bayesian models at different levels of detail are used in conjunction with measures of value of information and application of expected utility are used to control the progression of a dialogue and to make decisions regarding the treatment of miscommunication [9].

The second issue is concerned with the types of dialogue that we are able to model. Currently our system models form-filling dialogues in which there is a fixed set of information to be elicited before the system can bring the transaction to a close. However, we believe that our architecture is in principle capable of being extended to cope with more open-ended dialogues by generalizing the representation of the DialogFrame and by including mechanisms that enable the system to generate dialogue moves based on its current state without the constraints of slot-filling. This approach is similar to that of the Trindi Dialogue Move Engine [8], except that we would also retain the benefits of our object-based architecture

7. References

[1] Clark, H.H.. *Using Language*. Cambridge University Press, Cambridge, 1996.

[2] Traum, D.R. Computational Models of Grounding in Collaborative Systems. *AAAI Fall Symposium on Psychological Models of Communication*, 124-131, November, 1999.

[3] http://communicator.colorado.edu

[4] O'Neill, I.M., Hanna, P., Liu, X., McTear, M.F., "The Queen's Communicator: An Object-oriented Dialogue Manager", *Proceedings of EuroSpeech2003.* Geneva, September 2003.

[5] O'Neill, I.M., McTear, M.F., "Object-Oriented Modelling of Spoken Language Dialogue Systems", *Natural Language Engineering* 6 (3-4), 341-362, Cambridge University Press, 2000.

[6] O'Neill, I.M., McTear, M.F., "A Pragmatic Confirmation Mechanism for an Object-Based Spoken Dialogue Manager", *Proceedings of ICSLP-2002*, Vol. 3, 2045-2048. Denver, September 2002.

[7] Heisterkamp, P., McGlashan, S., "Units of Dialogue Management: An Example", *ICSLP96 - Proceedings of the Fourth International Conference on Spoken Language Processing*: 200-203, Philadephia, 1996.

[8] Larsson, S., Traum, D.R. Information state and dialogue management in the TRINDI Dialogue Move Engine Toolkit. *Natural Language Engineering* 6(3-4):323-340, 2000.

[9] Paek, T. & Horvitz, E. Conversation as action under uncertainty. *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, 445-464. Morgan Kaufmann, 2000.