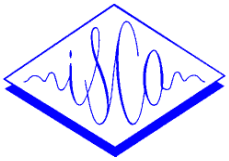

7th International Conference on Spoken Language Processing [ICSLP2002]
Denver, Colorado, USA
September 16-20, 2002

A PRAGMATIC CONFIRMATION MECHANISM FOR AN OBJECT-BASED SPOKEN DIALOGUE MANAGER

Ian M. O'Neill[1] *and Michael F. McTear*[2]

[1]School of Computer Science, The Queen's University of Belfast,
Belfast, BT7 1NN, N. Ireland.
i.oneill@qub.ac.uk

[2]Faculty of Informatics, University of Ulster,
Newtownabbey, Co. Antrim,BT37 0QB, N.Ireland
mf.mctear@ulst.ac.uk

ABSTRACT

Using a relatively simple confirmation strategy based on confirmation statuses, discourse pegs and request templates it is possible to implement an effective mixed initiative, natural language dialogue system. If the basic confirmation strategy is inherited by a set of specialist domain objects, the approach becomes particularly useful, allowing dialogues in several business areas to be conducted in a single implementation. The following paper outlines the basic confirmation mechanism and illustrates how this can be augmented by 'rules of thumb' that are encapsulated in implemented domain experts and that recommend particular strategies for continuing a transaction, given particular combinations of confirmed and unconfirmed information.

1. INTRODUCTION

Object-orientation provides an intuitive separation of inheritable generic functionality on the one hand and on the other hand domain-specific, specialized functionality that is supported by the generic elements of the system. When applied to the area of natural language dialogue, this enables the developer to create a generic, automated dialogue confirmation strategy, based on confirmation statuses and discourse pegs and guided by a request template of feature-value pairs that are used to complete a particular transaction. The particular request template used in a given type of transaction can be encapsulated in one of several domain expert classes, where the ideal combination of information required to complete a particular transaction can be supplemented with various rules of thumb that will prompt the user for significant missing information, taking into account typical combinations of information that may already have been confirmed.

Thus, while the generic confirmation strategy will ensure that information newly supplied by the user is confirmed, and information changed is reconfirmed, and so on, the nature of that information may differ significantly from domain to domain. Likewise the system may respond to confirmed information in quite different ways depending on the domain – as it either completes a domain-specific transaction or attempts to elicit important missing information from the user.

2. THE CONFIRMATION STRATEGY

2.1. The basic data structures

In each of the specialist domains in which it operates the system uses a structure containing a list of appropriate feature value pairs to store and monitor the information supplied by the user. In our implementation of the dialogue manager [1] these structures take the following form:

concept(concept type, attribute list) (1)

Each member of the *attribute list* in the *concept* construct takes the form:

attribute(attribute type, attribute value, confirmation status, system intention). (2)

The implemented specialist classes that deal with particular kinds of transaction instantiate the *concept type* and the *attribute type* with appropriate values. The following are instantiated examples of concept and attribute terms such as would be used by a class specializing in travel enquiries.

```
concept(travel_booking,
        Attribute_list).                    (3)

attrib(from,belfast,
       new_for_system,confirm).             (4)
```

The third and fourth arguments in this attribute structure (the *confirmation status* of the *attribute value* and the *system intention* concerning it) will be discussed in more detail presently.

The list holds as many attributes as are potentially relevant to the type of transaction. As *attribute value*s are supplied and confirmed by the user, these are compared against the request templates for the particular business domain. The system adopts a slot-filling approach, permitting the user over- and under-informative responses. The request templates indicate which *attribute value*s are required to complete particular kinds of transaction, and which further actions should be taken given a combination of *attribute value*s that is in itself insufficient to complete the transaction. The request templates and their associated dialogue furthering strategies – e.g asking the user

for important missing information, or making inferences about missing information - are domain specific. However, the processes for assessing and assigning the *confirmation status* of a particular *attribute value*, as well as the mechanism that allows a request template to be consulted and its heuristic dialogue furthering strategies to be enacted, are entirely generic. In other words, regardless of the business domain, the patterns by which the confirmation statuses of *attribute value*s evolve, and the sequence in which the system prepares for its next move, are the same.

2.2. Interplay of generic and domain specific functionality

Figure 1 gives an overview of the main system components in the prototype dialogue system.

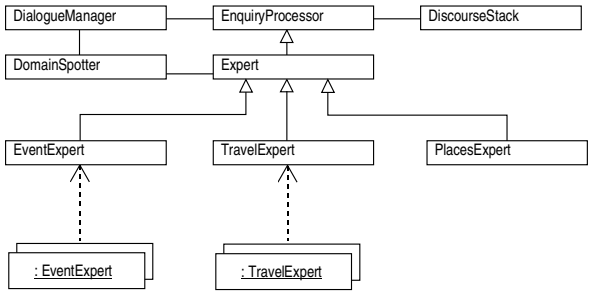

Figure 1*: Class Relationship Model: Overview.*

The domain-specific request templates, with their rules for completing a domain-related transaction or for eliciting further required information, reside in the appropriate Expert subclasses. The generic functionality for maintaining and modifying the confirmation status of the *attribute value*s used in the various request templates is encapsulated in the Enquiry Processor, from where it is inherited by the Expert subclasses. Moreover, each of the Expert subclasses may have one or more instances, corresponding to a specific real-world business. These instances apply the Enquiry Processor's generic dialogue management strategies, and the rules associated with the domain-specific request templates, to the specific data of their own service offering. The instance has at its disposal the richness of the functionality inherited via the object hierarchy.

2.3 Generic enquiry processing strategies

The generic Enquiry Processor has two strategies – one based on confirmation statuses, the other based on discourse pegs that are used in combination to help it decide whether the attributes of a user's utterance have been confirmed to a sufficient degree to be used as input in the final transaction (the actual process of reserving a ticket for a journey or an event, say).

Enquiry Processor assigns an appropriate status to each of the attributes in the user's utterance (from the set defined by Heisterkamp and McGlashan [2]) and updates the statuses as the dialogue evolves. Enquiry Processor is designed to perform this function regardless of how many attributes might be associated with the concept expressed in the user's utterance. Within Enquiry Processor the attributes are processed simply as members of a list of arbitrary length. Each attribute is structured as (2) above. The attribute's *confirmation status* is generally assigned one of the following values:

new for system, *inferred by system*, *repeated by user, modified by user*, *negated by user* (5)

In the prototype implementation a procedure called *evolve* represents the rules by which the statuses are updated as values are repeated, modified or negated by the user, or inferred by the system. *evolve* takes the following form:

evolve(type, last value, last status, current value, new value, new status, new intention). (6)

The *new status* of any given attribute is determined by its *current value* (i.e. its value in the user's current utterance), its *last value* and its *last status*, these last being retrieved by the Enquiry Processor from the Discourse Stack, a stack of concepts of the form shown in (1) above and representing the discourse history.

2.3.1 System Intentions

Enquiry Processor also contains the generic rules that determine the system's spoken response to an attribute, taking into account not only the status of the individual attribute but also of the other attributes in the overall enquiry concept. For example, in order to prioritize, and limit the number of, attributes that it refers back to the user for confirmation or further clarification, the system notes the number of responses and kinds of responses that it could potentially make to the different attribute statuses. A suite of *resolve* rules then determines the system's actual spoken response, such that the system deals firstly with individual attributes that have been negated, then individual attributes that have been modified, then up to three attributes that must simply be confirmed, and finally with attributes that must still be specified by the user. These upcoming system actions are the system 'intentions', and the various intention values are assigned to the relevant attributes in the discourse state prior to the next system utterance. The principal intentions are *confirm* (for new values), *repair confirm* (for a modified value), *repair request* (for a negated value) and *spec* (to have the user specify a required value). The 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.

The prioritizing of the responses and the limits on the number of attributes addressed in each kind of response, is an attempt on the one hand to recognize the possible significance of information that the user has negated or changed, and on the other hand to keep within user-friendly bounds the amount of information that the system attempts to elicit in a single dialogue turn. Pragmatically the system's intentions focus on its generic confirmation strategy of ensuring that the user explicitly or, by repetition, implicitly confirms the *attribute values* that he or she has supplied and that the system knows, by reference to its various request templates, to be relevant to a particular type of request. Once values are confirmed, the system is ready to process the transaction in full or in part using its domain-specific request templates.

The record of the *system intention* on a per attribute basis on the discourse stack is especially important in the event that the user replies simply 'yes' or 'no' to the system utterance, for

the system will 'remember' its purpose behind the utterance that provoked the 'yes' or 'no' response and will be able to work out what change in attributes' *confirmation status* the yes or no should bring about.

2.3.2 Using confirmed attribute values

Alongside the processing of the attributes' *confirmation status*es, each attribute also has a 'discourse peg' that is incremented by 1 when the user repeats a value, zeroed if the value is modified, set to -1 if the value is negated, and set to -2 if the attribute is deemed by the user to be superfluous to his or her enquiry (for example, the user might say that a return trip is not required, at which point the *return day* attribute of the *travel enquiry* concept will be set to -2). The aim here is to ensure that every attribute has been adequately confirmed (in the prototype its peg must simply be set to a value greater than zero) before it is used to complete a transaction.

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.

2.3.3 Agent-like behaviour

In general the system grounds the dialogue on tasks which it is programmed to manage, and it may for example explicitly ask the user "Is this a travel booking or an event booking?", when the user does not make his or her intentions clear. In this respect the system has more in common with the task-oriented approach of the SUNDIAL system [3], where the system makes clear to the user what *it* is trying to do, than with systems that additionally attempt to analyse the possibly complex and vaguely expressed intentionality of the user (e.g. TRIPS [4]). Nevertheless, the resulting dialogue manager, by attempting to confirm the content of the user's utterances, and use that information to fulfil the user's request as far as that is possible in the light of a specific service offering, manifests much of the behaviour of a system acting as 'agent' as discussed by Traum [5].

By 'knowing' in its suite of request templates and associated rules what information it requires and when and how it should elicit the information from the user, and by requiring implicit or explicit confirmation of the information before using it to further the transaction, the system's questioning of the user and its processing of the transaction details take on a superficially human quality. The impression to the user is of a person who knows how to perform particular tasks, but is careful that he or she has all the right details before doing so.

2.4 Detailed enquiry processing

Let us now examine in greater detail the manner in which the Enquiry Processor formulates the system's next intentions and the sequence in which confirmed information is used in conjunction with domain-specific request templates in order to complete or further the transaction.

Enquiry Processor takes as its input the user's current utterance. The following is a typical 'concept' representing an utterance made by the user. In this example the user has expressed a wish to fly to Dublin on Friday. The attributes that convey this information are in bold italics – the attribute list is enclosed between the outermost pair of square brackets:

```
concept(travel_booking,[
attrib(from,[],Status1,Intention1),
attrib(to,dub,Status2,Intention2),
attrib(day,fri,Status3,Intention3),
...,
attrib(mode,airline,Status6,Intention6),
...]).
```

Enquiry Processor compares each attribute from the user's current utterance with each corresponding attribute in the last discourse state on the discourse stack. Where an attribute value has changed, the system uses its *evolve* procedure (6) to assign one of the available statuses (5). Enquiry Processor then uses a *resolve* procedure to prioritize, and limit the number of, attributes that it refers back to the user for confirmation or further clarification.

The following is a typical discourse state for the travel domain, the result of the system having processed the user's utterance – in which previously uttered information has been repeated. In this case the system has formed the intention to confirm an inference of its own – which it made as a result of a rule associated with a domain-specific request template – that the user will travel from Belfast (attribute in bold italics).

```
concept(travel_booking,[
attrib(from,bfs,inferred_by_sys,
       confirm),
attrib(to,dub,repeated_by_user,blank),
attrib(day,fri,repeated_by_user,blank),
...,
attrib(mode,airline,repeated_by_user,
       blank), ...]).
```

In its *new response* the system formulates its utterance based on its intentions – in this case it will attempt to confirm its inference.

Thus, at a generic level , the system is able to monitor and react to new information and changes in information supplied by the user. Additionally Enquiry Expert indicates the circumstances in which a discourse peg for a particular attribute should be adjusted. However, it is the Expert subclass (an Event Expert or a Travel Expert in the prototype implementation) which performs any required operation on the discourse pegs, which are among the subclass's own data: the discourse pegs are domain-specific, since each peg is associated with one of the several attributes involved in a domain-related inquiry and is used to monitor the level of 'confirmedness' of that particular attribute. Above a particular 'confirmedness threshold', information supplied by the user may be used in different combinations either to complete the transaction, or to work out alternatives, in accordance with the request template and associated rules encapsulated in the Expert subclass. Furthermore as it attempts to complete a transaction, the system uses data associated with specific instances of the Expert subclass, analogous to using the schedule information of a particular airline or the events listing of a particular theatre, for example.

The following are examples of the sorts of algorithms that are used to complete or further a transaction given a set of confirmed pieces of information (attributes whose discourse

pegs are set greater than zero in the prototype implementation). Two flavours of algorithm were used in the prototype:

- *template checks*, which either conclude a transaction or advise the user to supply further information that can be used to complete the transaction , and
- *remedial checks*, which given confirmed but invalid combinations of information – a requested day might not be available in the schedule of a particular travel expert instance, for example – can advise an alternative, valid request.

In each case, the advice given is based on the sort of 'rule of thumb' or heuristic that a human advisor might use. Encoded at the expert subclass level, the rules can be as extensive as is necessary to simulate typical dialogue behaviour of a co-operative human agent in the particular business domain – i.e. rules included in the Travel Expert class are rules that might be used by any travel agent, regardless of the instance-specific service information that is consulted.

In the current relatively simple prototype, if the system has confirmed the place of departure, the destination, the day of departure and the departure time (that is, their discourse pegs are set greater than zero), 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 > 0
   AND
     the Handling Agent's schedule
    includes a service for
    departure point, destination,
    day and departure time)
THEN
    instruct the Dialogue Manager
    to generate a final system
    utterance confirming a
    reservation for
    departure point, destination,
    day and departure time.
```

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.

Should the *template check* fail - because the details supplied by the user and confirmed by the system prove to be an invalid combination in terms of the handling instance's database - then Enquiry Processor will move on from the *template check* to invoke the Expert subclass's remedial *check*s in an attempt to find an alternative service for the user. For example, if the flight does not depart at the time the user requested, the system might be able to use the instance's data to suggest another time. Again, in more structured form, processing for a typical Travel Expert's *check* can be represented as follows:

```
IF
   (the discourse pegs for
    departure point, destination,
    day and departure time are > 0
   AND
    the Handling Agent's schedule
    DOES NOT include a service for
    departure point, destination,
    day and departure time
   AND the Handling Agent's schedule
    includes a service for
    departure point, destination, day and
    another departure time)
THEN
   instruct the Dialogue Manager
   to generate an utterance suggesting
   another departure time.
```

In the present implementation the system will continue to seek information until it has confirmed enough values to conclude the enquiry, or until the user quits.

3. CONCLUSIONS

The proposed object-oriented architecture has the significant advantage that it allows a generic, automated confirmation strategy to be inherited by implemented domain experts that operate in different business areas. The generic confirmation strategy involves monitoring the confirmation status of the individual attributes associated with a particular kind of transaction, and allows the system to use a simple system of priorities to decide which attributes, and how many, need to be confirmed or re-confirmed with the user in its next utterance. A discourse stack not only provides a history against which confirmation status can be evolved but also allows the system to 'remember' its own intentions and so interpret 'yes/no' responses from the user. Finally, suites of template checks and remedial checks at the level of domain expert subclasses enable the system to recreate some of the transaction guiding advice typical of a human domain expert.

4. REFERENCES

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

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

[3] Youd, N. and McGlashan, G. S. *Semantic Interpretation in Dialogue*, University of Surrey, 1992.

[4] Allen, J., Byron, D., Dzikovska, M., Ferguson, G., Galescu, L., Stent, A., "Towards conversational human-computer interaction", *AI Magazine*, 2001.

[5] Traum, D., "Speech Acts for Dialogue Agents", *Foundations of Rational Agency*: 169 – 201, Wooldridge, M. and Rao, A. (eds.), Kluwer, 1999.