Proceedings of EACL '99

An Object-Oriented Approach to the Design of Dialogue Management Functionality

Ian M. O'Neill and **Michael F. McTear**
Faculty of Informatics
University of Ulster
Newtownabbey
Co. Antrim
BT37 0QB
N. IRELAND
mf.mctear@ulst.ac.uk

Abstract

Dialogues may be seen as comprising commonplace routines on the one hand and specialized, task-specific interactions on the other. Object-orientation is an established means of separating the generic from the specialized. The system under discussion combines this object-oriented approach with a self-organizing, mixed-initiative dialogue strategy, raising the possibility of dialogue systems that can be assembled from ready-made components and tailored, specialized components.

1 Introduction

For the purpose of developing automated systems, dialogues may be seen as comprising commonplace routines on the one hand and specialized, task-specific interactions on the other. In software engineering, object-orientation has proved to be an effective means of separating the generic from the specialized, and more particularly, of letting the specialized inherit the generic (Rumbaugh *et al.*, 1991). Identifying inheritable generic functionality (for confirmation, repair of misunderstanding, personalization of utterances, etc.) and specialized or highly domain-specific functionality, opens the way to dialogue systems that can be assembled largely from ready-made components and extended with the addition of more specialized components. The prototype system that we have been developing in Prolog++ for the last year combines this familiar object-oriented approach with a self-organizing, mixed-initiative dialogue strategy. Pseudocode is used here to represent the Prolog processing.

2 Identifying the generic and the specialized

In the course of developing the prototype system a number of important generic elements have been identified that can be ported with a minimum of alteration between domains. These generic elements are now introduced.

2.1 Dialogue Manager

In any system that is concerned with conducting a dialogue with a user, a mechanism is required for receiving, forwarding for processing, and outputting semantic contents of utterances. This responsibility falls to a Dialogue Manager.

2.2 Domain Spotter

Any system that is intended to handle processing across a number of real-world areas of expertise requires a means of associating key semantic content of the user's utterances with one or more of the available domains. This responsibility falls to the Domain Spotter.

2.3 Discourse Stack

Any system dealing with a transaction that involves multiple dialogue turns must have a means of logging a) what it believes the user has said, b) the degree of 'confirmedness' of what has been said, and c) how the system has decided to respond. Maintaining a record of the evolving discourse, and providing the means of creating and retrieving entries for individual utterances, are the responsibilities of the Discourse Stack.

2.4 Enquiry Processor

Given the current difficulties of speech recognition, and the possibility that a user will misunderstand or change his or her mind, any system conducting a complex transaction must have a strategy for confirming the semantic contents of the user's utterances and for proceeding with the transaction only when details have been adequately confirmed. The current system increments or decrements levels of 'confirmedness' depending on whether the user repeats or confirms, alters or negates values. If necessary, the system queries the user explicitly about values that are new, altered or negated. The responsibility for these purely generic, mechanistic confirmation routines falls to the Enquiry Processor, whose strategies are inherited, via a generic agent or Expert, by subclasses that have their own domain-specific processing heuristics.

2.5 Expert

Each of the more specialized agents within the system must have access to wider system resources and have ways of providing the wider system with high level information about its processing abilities. Supporting these common behaviours and characteristics is the responsibility of the generic Expert class.

Other parts of the system must be tailored to represent the specialized knowledge and processing abilities of real-world human specialists. These are introduced next.

2.6 Expert Subclasses

For each business area within the system there must be functionality a) to decide what information to elicit next, or what information to infer, given that certain information may already have been provided, b) to check the validity of the combinations of information provided, c) to give the most helpful guidance when the user is having difficulty completing the enquiry, and d) to decide when sufficient, confirmed information has been provided to conclude the transaction. Such functionality is specific to the Expert subclasses within the system, and recreates in sometimes quite extensive sets of domain-specific heuristics, the kind of behaviour (e.g. 'if details for an outward journey are received, check if a return is needed'; 'if a venue has been confirmed but not a day, ask for the day') that would characterize any human expert in a particular business domain - a travel agent or a theatre booking clerk, for instance. The current subclasses are Travel Expert, Event Expert and Place Expert.

2.7 Expert Instances

The system must contain detailed service information of the kind that in the real world is associated with individual businesses. Businesses are represented by instances of Expert subclasses. The instances represent particular airlines, railways, theatres, cinemas and so on; they have access to the data - concrete schedules and timetables - that must be consulted if a transaction is to be meaningful.

3 Some important system characteristics

The current prototype (Figure 1 below) focuses on dialogue management. It is not intended to transcribe and parse raw spoken input, nor compose complete utterances for speech generation. Rather, the system accepts an input of concepts and attributes in the form *concept(action type(attribute list))* and outputs concepts and attributes in similar fashion.

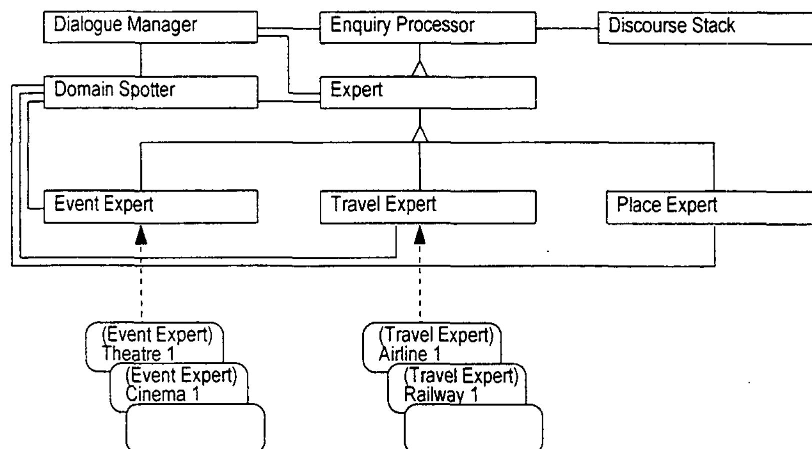

Figure 1. Main System Components.

If the system is to conduct a dialogue as a human interlocutor might, it must use to best advantage whatever information it is given - whether that information was explicitly sought or not - and then be able to ask for information it still requires. Such self-organizing behaviour, as opposed to simpler state transitions (Novick and Sutton, 1996), generally has one of a number of possible motivations. The system may be plan-based, attempting to identify and understand the ramifications of the problem the user wants to solve (Allen *et al.*, 1996). Alternatively it may attempt to prove theorems, questioning the user for the missing facts that it needs to know in order to help him or her complete some complex task (Smith and Hipp, 1994). Or the system may attempt to identify or elicit specifically those facts that it needs to complete a 'request template' for a particular transaction.

It is essentially this last approach that the current prototype has adopted, and to this extent it resembles the SpeechMania system developed by Philips (Aust and Oerder, 1995), which has already been used successfully to implement a speech-based timetable enquiry system for Swiss Federal Railways (Aust *et al.*, 1995). However, by additionally identifying generic and specialized functionality, including heuristics that would characterize a human expert, it becomes possible to create a dialogue management system that can cope with several real-world enquiry domains, or a number of complex subtasks, in one and the same adaptable, extensible implementation.

4 Generic behaviour - domain-specific knowledge

The system is coloured throughout by a design philosophy that keeps the higher-level system components largely ignorant of the capabilities of the lower-level system. This has the advantage that higher-level, generic dialogue functionality can remain unchanged as the lower-level system is adapted for specialized real-world application areas. However, it goes without saying that the higher-level components must know how to access the lower-level functionality.

Domain Spotter is one such higher-level component in the current prototype. Its purpose is a very simple one: it consists of a collection of rules that the Dialogue Manager uses to pass enquiries of different types to the most appropriate domain experts. For it to work - in the current implementation - Domain Spotter relies on the assumption that recognizer-grammar functionality (outside the scope of the current implementation) will be sufficiently powerful to identify key semantic content from the user's utterance, content that may be characteristic of possibly one or more real-world business domains. Domain Spotter's heuristics then tell it broadly where the corresponding domain-related functionality resides in the system. It may then have to determine, if necessary by quizzing the user further, which of several Expert subclasses is best suited to the current enquiry.

If, for example the user's utterance indicates simply that he or she wants to make a booking and no further details are given, Domain Spotter is programmed to interrogate the Expert subclasses to find out which ones can handle bookings. (Prolog++ conveniently provides a call to all subclasses of a given type.) On the basis of the responses it obtains, it may subsequently have to ask the user to narrow the scope of the enquiry. For the moment, however, if a subclass *does* handle bookings, it will simply push its *class area* attribute (indicating its area of competency: *travel*, or *events*, say) on to the *class candidate list* within Domain Spotter. Otherwise it performs a Prolog cut and allows the call to pass to another subclass. In the next dialogue turn the Dialogue Manager uses the contents of the list to offer the user a selection of business areas to choose from. Figure 2 below (with simplified calls) illustrates the process.

If the user's enquiry is more specific - 'I'd like to book a trip on Friday' or 'I'd like to make a theatre reservation' - such that travel- or event-related semantic content might be readily

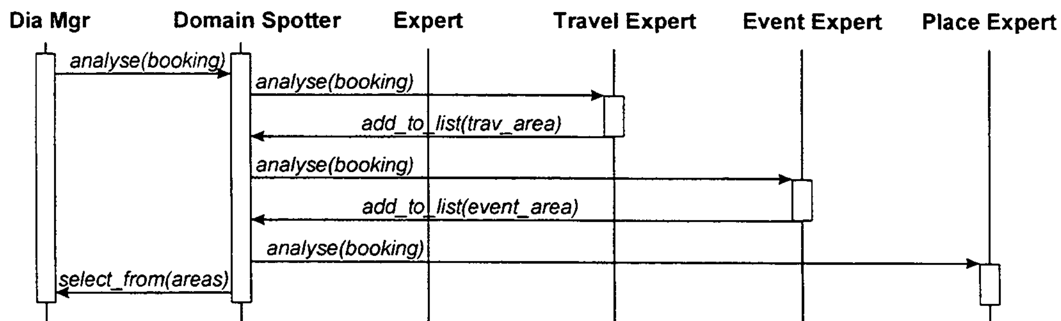

Figure 2. Finding the Relevant Subclass

identified by a recognizer-grammar - Domain Spotter, in its high-level *analysis*, performs like a human receptionist or operator and passes the enquiry to the most relevant subclass for a more detailed *analysis* specific to that subclass.

Any available attributes of the travel enquiry - day, time, etc. - are also forwarded to the specialized domain expert. The expert then has to decide using its own heuristics what use it can make of the attributes.

5 Finding an object to handle the transaction

At this point the enquiry is still being processed quite generically at the level of an Expert subclass - let us assume the Travel Expert, in order to explore further the evolution of a typical transaction. However, for the enquiry to stand any chance of reaching a successful conclusion, it must eventually be processed by an instance of a class (in object-oriented terms a specific 'object'), representing an actual company or organisation that has a highly detailed knowledge of the required service. (Cf. Wang (1998), who uses a semantic grammar in a base class to provide high level understanding of an utterance, and then finds a 'best match' from among the grammars of derived classes for a more detailed understanding.)

Thus, having been passed the enquiry by Domain Spotter, the Travel Expert subclass now attempts to identify the most suitable Travel Expert instance to handle the enquiry, or if it is unable to do so in this dialogue turn, to elicit further information from the user to help it identify a 'handling instance'. In a move analogous to the one adopted by Domain Spotter previously, the Travel Expert interrogates its instances and has them push their area of expertise (their *area* attribute - *railway*, *airline*, etc.) on to Domain Spotter's *candidate list*. In the next turn the Dialogue Manager will ask the user to narrow the enquiry to one of the *areas* available.

Although the system may request specific information (as in the turn above), the user may supply rather more than this. Using the heuristics of the relevant Expert subclass (here the Travel Expert), the system analyses the supplied information, to try to establish the context of the transaction, and then to process the transaction within that context. Again, the system is aiming to find the object (the representation of a real-world business) that is best suited to processing the transaction to its conclusion. Let us explore this further.

6 A flexible response

At the early stages of the transaction Domain Spotter polls the Expert class and subclasses (on the basis of the semantic content of the user's utterance) with the goal of finding a handling instance. If in response to the system's question 'Is that a railway ticket or an airline ticket?' the user says that he or she wants a ticket with a particular airline, processing is immediately taken up by the appropriate airline instance. Alternatively the user might respond along the lines that he or she wants 'a plane ticket for London on Friday at around nine a.m.' Assuming that a phrase such as 'ticket to London' has been successfully parsed as a travel-related request, Domain Spotter will pass the query to the Travel Expert class, which in turn will interrogate its instances to see how many have *airline* as an attribute and travel to the destination on the day and at the time requested. Figure 3 below illustrates the process. If the instance is unable to meet the criteria it simply performs a cut and passes the call to the next instance. Any instance that can provide the required service adds its

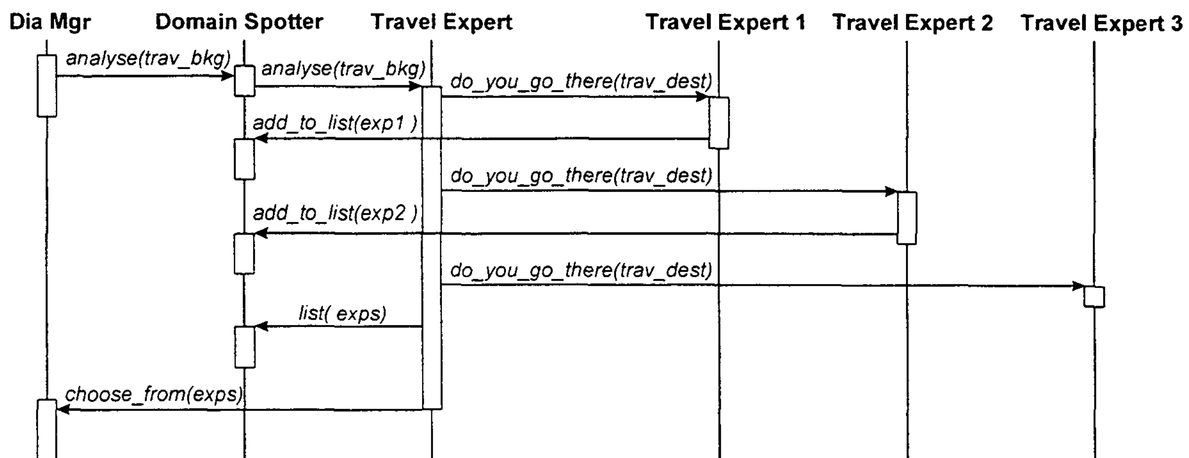

Figure 3. Identifying Appropriate Instances

mnemonic, its unique identifying name, to Domain Spotter's *candidate list*. Again, the analogy with Domain Spotter's own interrogation technique holds good.

Now the role of the instances becomes more important. In the prototype system the instances contain, as one of their attributes, specific details of the service they offer: in the case of a Travel Expert instance this will be a schedule; in the case of an Event Expert instance a programme of shows. In a more realistic implementation the instance is more likely to serve as the gateway to a corporate database. Nonetheless, whatever the implementation, the instance will serve as the means by which the system at large has access to the detailed information it needs to complete the transaction.

If as a result of the interrogation above, there are several candidates for 'handling instance' on Domain Spotter's *candidate list*, the Dialogue Manager, in the system's next turn, will prompt the user to choose one of them (and, of course, accept any additional information that the user might provide). If there is only one candidate, or indeed if at some point the user specifically names the instance he or she wants to provide the service ('I'd like to book a flight with Aer Lingus.'), the system can move the dialogue into its final stage, where the semantic content of the user's utterances is methodically confirmed and checked for compatibility with the instance's data, and where data still required for closure of the transaction are elicited from the user.

7 An engine for confirmation strategies

Perhaps the most domain-independent element of the system is the Enquiry Processor class, which implements the generic confirmation strategies that must be performed in a system intended to cope with imperfect speech recognition, and users who change their mind. In reality, Enquiry Processor adopts quite a mechanistic approach to confirmation and this routine functionality is inherited, via the Expert class and the Expert subclasses, by the 'handling instances' that ultimately process the enquiry.

Enquiry Processor has two strategies, 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). On the one hand, Enquiry Processor assigns an appropriate status to each of the attributes in the user's utterance (from the set defined by Heisterkamp and McGlashan (1996)) 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 - though realistically even a complex concept, such as a booking for a return trip, will have no more than about fourteen attributes, covering place of departure, destination, details of outward and return journey, and so on. Within Enquiry Processor the attributes are processed simply as members of a list of arbitrary length. Each attribute is structured as follows.

attribute(type, value, status, system intention)

The attribute's *status* is generally assigned one of the following values:

- *new for system*
- *inferred by system*
- *repeated by user*
- *modified by user*
- *negated by user*

A suite of *evolve* predicates represent 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).

The *new status* of any given attribute is therefore determined by its *last value*, its *current value* (i.e. its value in the user's current utterance), and its *last status*. The *last value* and *last status* are taken from the Discourse Stack, a discrete system component comprising a list and functionality to push and pop the concepts and attributes that document the user's utterances and the system's responses. Enquiry Processor also contains the 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. Following invocation of the rules, the *system intention* parameter of the *attribute* term is set to the system's intended next move in regard to the attribute - whether it will confirm, query, etc., the attribute. This is especially important in the event that the user replies simply 'yes' or 'no' in his or her subsequent turn. Moreover, Enquiry Processor's rules not only determine the system's responses, but also help it prioritize its responses: for example, before doing anything else it will question the user about any value that he or she has negated - since negation represents a significant misunderstanding or change of plan; if it needs to confirm attributes, it will attempt to

confirm no more than three in a single turn. Its set of priorities permitting, the system will perform a *repair request* on a negated value, a *repair confirm* on a modified value, a *confirm* on a value that is new to the system or has been inferred by the system, and a *spec* on any value that requires the user to choose between one of several options (Heisterkamp and McGlashan, 1996).

Alongside this processing of the attributes' statuses, each attribute has a 'discourse peg' that is incremented by 1 when the user repeats a value, zeroed if the value is modified, and set to -1 if the value is negated. The aim here is to ensure that every attribute has been adequately confirmed (in this prototype its peg must simply be set to a value greater than zero) before it is used to complete a transaction.

8 Knowing when enough is enough

As well as implementing these mechanisms for evolving attributes' statuses and determining the system's next utterance, the Enquiry Processor class has a mechanism, a *template check*, for deciding whether the user has supplied enough information to complete the transaction.

Enquiry Processor's functions are performed in the context of a specific handling instance, so the *template check* uses the data that are encapsulated in the current handling instance. Again, in an actual real-world system these data might be contained in the database to which the instance has, from the overall system's perspective, exclusive access. For each Expert subclass there are normally a number of different potential combinations of confirmed data that can be used to successfully conclude a transaction: collectively these constitute the 'request template' for the subclass. The request template additionally indicates information that the system should prompt for next, given a particular combination of data that have already been confirmed.

Thus, for example, 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, 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 processing for *template check* 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 check on the template 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 perform a number of remedial *checks*. These *checks* again use heuristics that are valid at the level of the Expert subclass, in combination with data that are specific to the handling instance. In performing its *checks* Enquiry Processor aims to offer the user an alternative course of action - 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 *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.

9 The way ahead

Currently the system is being tested in a selection of short travel- and event-related transactions, during which it processes *concept* terms whose attributes the user may alter or negate to simulate misrecognition and/or revised requirements. Its performance has been accurate and its responses near-instantaneous on a 100 MHz Pentium PC with 16 MB RAM running under Windows 95. Typically the test transactions require the user and the system each to make between three and seven utterances.

The prototype system has recently been amended to allow the confirmation strategy to come into play as soon as the user has supplied a *concept* with confirmable attributes - even before a handling agent has been identified. With the confirmation strategy now being introduced earlier, and potentially having to deal with more amendments, negations and additional comments by the user, further investigation will be required to determine the best way to prioritize and meaningfully group the attributes that the system has to query for different enquiry types. It is likely that additional heuristics will be required at the Expert subclass level.

Peer objects will also be developed to work alongside the current Expert subclasses, providing highly specialized but essentially domain-independent functionality - such as processing credit card details or gathering address information. The aim is to create a suite of components which, with their encapsulated real-world expertise, can be combined 'off-the-shelf' for functionally rich dialogue management. The object architecture readily supports the addition of still more specialized subclasses and instances as further functionality is required.

References

Aust, Harald, Martin Oerder, Frank Seide, and Volker Steinbiss. 1995. The Philips automatic train timetable information system. *Speech Communication* 17: 249-262.

Allen, James F., Bradford W. Miller, Eric K. Ringger, and Teresa Sikorski. 1996. A Robust System for Natural Spoken Dialogue. *Proceedings of the 34th Annual Meeting of the ACL*: 62-70.

Aust, Harald and Martin Oerder. 1995. Dialogue Control in Automatic Enquiry Systems. *ECSA Workshop on Spoken Dialogue Systems*: 121-124.

Heisterkamp, Paul and Scott McGlashan. 1996. Units of Dialogue Management: An Example. *ICSLP96 - Proceedings of the Fourth International Conference on Spoken Language Processing*: 200-203.

Novick, David G. and Stephen Sutton. 1996. Building on Experience: Managing Spoken Interaction through Library Subdialogues. *Proceedings of TWLT 11 - Dialogue Management in Natural Language Systems*: 51-60.

Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. 1991. *Object-Oriented Modeling and Design.* Englewood Cliffs, New Jersey: Prentice-Hall.

Smith, Ronnie W. and D. Richard Hipp. 1994. *Spoken Natural Language Dialog Systems: A Practical Approach.* New York: Oxford University Press.

Wang, Kuansan. 1998. An Event-Driven Model for Dialogue Systems. *ICSLP98 - Proceedings of the Fifth International Conference on Spoken Language Processing*: 393-396.