Available online at www.sciencedirect.com

SCIENCE @ DIRECT®

Speech Communication 45 (2005) 249–269

SPEECH COMMUNICATION

www.elsevier.com/locate/specom

Handling errors and determining confirmation strategies—An object-based approach ☆

Michael McTear [a,*], Ian O'Neill [b], Philip Hanna [b], Xingkun Liu [b]

[a] *School of Computing and Mathematics, University of Ulster, Shore Road, Newtonabbey, Northern Ireland BT37 0QB, UK*
[b] *School of Computer Science, Queens University Belfast, Northern Ireland, UK*

Received 9 February 2004; received in revised form 8 November 2004; accepted 11 November 2004

Abstract

A number of different approaches have been applied to the treatment of errors in spoken dialogue systems, including careful design to prevent potential errors, methods for on-line error detection, and error recovery when errors have occurred and have been detected. The approach to error handling presented here is premised on the theory of grounding, in which it is assumed that errors cannot be avoided in spoken dialogue and that it is more useful to focus on methods for determining what information needs to be grounded within a dialogue and how this grounding should be achieved. An object-based architecture is presented that incorporates generic confirmation strategies in combination with domain-specific heuristics that together contribute to determining the system's confirmation strategies when attempting to complete a transaction. The system makes use of a representation of the system's information state as it conducts a transaction along with discourse pegs that are used to determine whether values have been sufficiently confirmed for a transaction to be concluded. An empirical evaluation of the system is presented along with a discussion of the advantages of the object-based approach for error handling.

Keywords: Object-based dialogue management; Grounding; Information state; Error handling

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. 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

☆ This paper is an extended version of a paper presented at the ISCA tutorial and research workshop on Error Handling in Spoken Dialogue Systems, Château-d'Oex, Switzerland, 28–31 August 2003.

* Corresponding author. Tel.: +44 28 9036 8166; fax: +44 28 9036 6068.

E-mail addresses: mf.mctear@ulster.ac.uk (M. McTear), i.oneill@qub.ac.uk (I. O'Neill), p.hanna@qub.ac.uk (P. Hanna), xingkun.liu@qub.ac.uk (X. Liu).

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 (Clark, 1996; Traum, 1999). 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* (Larsson and Traum, 2000), 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 reviews current approaches to error detection and error correction in spoken dialogue. Following this we introduce an object-based approach to error handling and describe the representations that are used in this approach to manage the system's assumptions about the dialogue—in particular, the extent to which the interaction with the user is proceeding successfully. We then show how our system uses these representations to determine what sorts of confirmations should be employed at various stages within a dialogue and present the results of an empirical evaluation of the system. We conclude with a discussion of the advantages of the object-based approach and of some ideas for future work.

2. Approaches to error handling

Error handling for spoken dialogue systems has traditionally been viewed in terms of a number of stages that include error prevention, error detection, and error recovery (Skantze, 2003; Turunen and Hakulinen, 2001).

Error prevention occurs at the design stage, where the designer predicts potential errors and designs the system so as to prevent these errors from occurring. For example, the careful design of prompts may help to constrain the user's input to words that the system will be able to understand, while recognition grammar design can ensure that words that are easily confused and words that are frequently misrecognised are avoided. Similarly dialogues can be designed to direct the user to the sorts of input that the system can handle, and, where problems occur, appropriate use of verification strategies can help in recovery from the errors.

Error detection involves monitoring the ongoing dialogue for indicators of problems. Error detection can take place at the point where an error occurs (early detection), or at a later point in the dialogue, for example, after the system has attempted to verify its understanding of the user's input (late detection) (Skantze, 2003). The system's monitoring of the progress of the ongoing dialogue may also lead it to predict potential errors and to adjust its behaviour accordingly (error prediction).

2.1. Early detection of errors

With early error detection the system detects that something is wrong in the user's current utterance and takes immediate steps to address the problem. Detection of errors has focussed mainly on speech recognition errors, where the most commonly used measure for error detection is the acoustic confidence score (Komatani and Kawahara, 2000; Hazen et al., 2000). However, acoustic confidence scores are not entirely reliable, as there is no one-to-one mapping between low confidence scores and recognition errors, nor between high confidence scores and correct recognitions (Bouwman et al., 1999). A more reliable measure uses secondary properties of the decoding process, such as language model backoff patterns and information in the word-hypothesis lattice, to assess recognition confidence (Wessel et al., 1998; Evermann and Woodland, 2000). Other approaches include the use of features that can be monitored automatically within the dialogue and that might indicate whether a speech recognition hypothesis is likely to be correct. For example, Litman et al. (2000) have investigated the use of prosodic cues to predict automatic speech recognition performance, finding significant prosodic differences between correctly and incorrectly recognised utterances, while Walker et al. (2000a) found that NLP features, such as parsing confidence, recognised task, amount of context shift, and salience, were a reliable indicator of errors in spoken language understanding.

2.2. Late detection of errors

Most work on late detection of errors has examined how the user responds to the system's attempt to verify the user's previous utterance, although of course the detection of the error could also occur later than this in the dialogue. Krahmer et al. (2001) distinguished between positive and negative signals, finding that users employed negative signals, such as longer turns, marked word order, corrections, and no new information, more often when the preceding system utterance contained a problem, particularly when this utterance was an implicit verification. In contrast, they found that positive signals, such as short turns, unmarked word order, confirmations, and new information, were used when there was no problem. Krahmer et al. also investigated the predictive capacity of these cues, looking at whether it would be possible to automatically decide whether a preceding system utterance contained an error. They conducted a series of machine learning experiments in which they used features to be found in the system's verification utterance as well as the positive and negative cues in the user's response. They found that the best results (almost 97% accuracy) were obtained when all the features were used in combination. In a companion study they also examined prosodic information in the user's utterance, finding that user utterances that corrected a system's verification often contained a high boundary tone, high pitch, as well as a relatively long delay between the end of the system's question and the beginning of the user's answer (Krahmer et al., 2001). Similar results were reported in a study of American English human–machine dialogues (Swerts et al., 2000).

2.3. Error prediction

Error prediction is concerned with predicting potential problems in a dialogue based on features monitored in the dialogue so far. Once such problems have been detected, steps can be taken to prevent the errors, for example, by changing the dialogue strategy.

Litman et al. (1999) applied machine learning techniques to develop a classifier that could detect dialogues with poor speech recogniser performance. The features included acoustic confidence scores as well as features concerning dialogue efficiency and quality. The best classification involved the use of all the available features. In this study the features were collected over the complete dialogues and so the rules learned by the classifier could not be applied within the dialogues during runtime.

In a subsequent study Walker et al. (2000b) used a corpus of 4774 dialogues from the HMIHY (How May I help you) corpus to train the system to be able to predict problems that were likely to occur in a dialogue. In this case the aim was to make the predictions early on so that adaptation could occur within the ongoing dialogue. Predictions were based on information available to the system in the form of features extracted automatically from the log files of the dialogues. The results showed that rules could be learned to predict problematic dialogues using the fully automatic features with an accuracy ranging from 72% to 87%, depending on how much of the dialogue the system had seen so far. The features used included speech recognition features, such as confidence scores, utterance duration, and number of recognised words, as well as NLU (natural language understanding) features and dialogue features. The NLU features included confidence features for the parse (which involved assigning the utterance to one of the 15 tasks of the HMIHY system), as well as measures indicating inconsistency and context shift. Inconsistency was an intra-utterance measure that indicated discrepancy between tasks that the system appeared to be requesting, while context-shift was an inter-utterance measure that detected if the user's subsequent utterance was in conflict with a previous utterance, suggesting that the system's interpretation of either utterance may have been erroneous. Dialogue measures included a number of features concerned with prompts and confirmations, including keeping running tallies for these features.

Prediction accuracy was investigated based on the system having seen only the first exchange in the dialogue compared with predictions made after seeing the first two exchanges as well as identification accuracy after having seen the whole dialogue. Accuracy using the set of automatically available features was 72% after seeing just the first exchange, and after the second exchange it was 79%, while identification of problems after the complete dialogue was 87%. Thus the rules learned by the classifier to predict potential problems could be employed automatically in a dialogue system to contribute to improvements in system performance.

2.4. Error recovery

A spoken dialogue system can respond in several different ways to a user's utterance:

1. It can accept the utterance and continue the dialogue. This strategy may be used when the system is confident in its interpretation of the user's utterance, but carries the danger that the system may have misinterpreted the utterance. Misinterpretations are often only detected later in the dialogue, if at all (McRoy, 1998).
2. It can accept the utterance and attempt to verify it, either with an explicit or an implicit verification. This strategy gives the user an opportunity to reject the system's interpretation and to make a correction, resulting in late error detection.
3. It can reject the user's utterance and ask the user to repeat or re-phrase the utterance, or the system can re-start the dialogue. Rejections in current spoken dialogue systems usually occur when the acoustic confidence scores for the user's utterance fall below a given threshold.

The system's choice of strategy may also be erroneous. A false rejection, in which the system wrongly rejects a user's utterance, is less serious, as it leads to a request for repetition or confirmation, or a re-prompt. A false acceptance, in which the system adopts an interpretation of a user's utterance in which the confidence score exceeds the threshold is more problematic as it may not be easy for the user to correct the system and put the dialogue back on track.

There are several ways in which a spoken dialogue system can attempt error recovery. The most straightforward way to get the dialogue back on track is to simply ask the user to repeat or re-phrase the problematic utterance. Different verification strategies, based on different levels of acoustic confidence scores, have also been employed (Sturm et al., 1999). Another method, as suggested by Krahmer et al. (2001), is to use information available in the detection of the error, such as the different negative cues discussed earlier, and indicators such as heavier emphasis on the corrected word, to construct a more suitable follow-up question for the system to ask, rather than simply asking the user to repeat or confirm all of the information that had been elicited, especially when some of it was probably correct.

The most radical strategy for error recovery has been proposed by Skantze (2003), based on an analysis of human error handling strategies. In this study 40 dialogues were collected using a modified version of the Wizard of Oz method, in which the user's utterances were processed by a speech recogniser and passed to the operator (or Wizard) so that reactions to realistic speech recognition errors could be investigated. The domain was a direction-giving task in which both user and operator used landmarks and relative directions to solve the task. Different strategies were used by the operator following non-understandings. In addition to signals of non-understanding, as discussed above, other strategies were to continue with the route description or to ask a task-related question about the user's current location. Skantze proposes that these strategies are more natural as they avoid the traditional strategy of signaling non-understanding, which tends to lengthen the dialogue and make it appear problematic in the user's perception.

3. An object-based approach to error handling

The approach to error handling to be described in this paper is premised on the theory of grounding, in which participants in dialogue aim to establish common ground—i.e. the mutual belief that their dialogue utterances and obligations have been understood. At the same time participants follow the principle of least effort in seeking to minimise the costs of grounding. Thus while ensuring that all relevant information has been confirmed, an efficient dialogue manager is also motivated to avoid unnecessary grounding in the interests of shorter and more satisfying transactions.

This approach has been implemented in the Queen's Communicator (O'Neill and McTear, 2000, 2002; O'Neill et al., 2003). The system currently handles transactions in the domains of accommodation requests and bookings as well as event requests and bookings (theatre and cinema). The dialogue manager supports mixed initiative dialogue, in which the user can supply more information than requested by the system and in which the user can correct the system or change information that has already been elicited, if they so desire. A mixture of explicit and implicit confirmation strategies are used, depending on the current state of the dialogue, and the system may perform a combination of confirming and validating the elicited information as well as requesting additional information in a single turn. The system also supports domain switching, so that, for example, the user can switch to a request about cinema information while negotiating an accommodation dialogue. Thus the system must be able to identify the topic of the dialogue, determine what the user is requesting, and recover from errors if it has misrecognised the user's words or intentions. This is achieved through a process of appropriate grounding in which the system seeks to confirm its understanding of the user's words and intentions.

3.1. The system architecture

The architecture to support these functionalities leverages the strengths of object-orientation by separating generic behaviour from domain-specific behaviour. The system builds on the DARPA Communicator architecture, and specifically on the components developed within the CU Communicator project (Pellom et al., 2000). In the 'Queen's Communicator' system the dialogue management components were removed from the CU Communicator system and replaced by a Dialogue Manager (DM) implemented as a suite of Java classes. The DM interacts with a number of off-the-shelf components, including: for speech recognition the Microsoft English ASR Version 5 Engine as supplied with Windows XP; for semantic parsing the Phoenix parser (Ward, 1994); and for synthesised speech Festival (www.cstr.ed.ac.uk/projects/festival/). A phrase-based natural language generation module, developed at Queen's University Belfast, converts the DM's output concepts into well-formed natural language utterances that are passed to Festival for text-to-speech synthesis.

In order to enable mixed initiative interactions across domains, the system's behaviour is modelled as a collaboration between a cohort of implemented 'agents' (see Turunen and Hakulinen, 2001 for a similar approach). An agent is a specialist in a particular transactional area—e.g. booking accommodation or eliciting an address and uses its own domain-specific 'expert rules' to elicit information (e.g. information for making a hotel booking) that is stored in a specialised dialogue frame. Each agent thus encapsulates a skillset for a substantial dialogue or subdialogue. Fig. 1 presents the Expertise Hierarchy, comprising components that the system uses to implement its generic confirmation strategies (i.e. what to do when the user supplies new, modified or negated information), and its domain-specific, dialogue-furthering strategies (for example, what to ask for next in an accommodation enquiry, or a theatre reservation, or when eliciting credit card details). In our terminology the DialogManager represents the entire dialogue management system, while the DiscourseManager is specifically responsible for enabling the system to follow the evolution of the dialogue and maintain a consistent confirmation strategy.

The DialogManager co-ordinates dialogue turn-taking and has a suite of business domain experts, such as an AccommodationExpert and an EventExpert. The DialogManager also has a

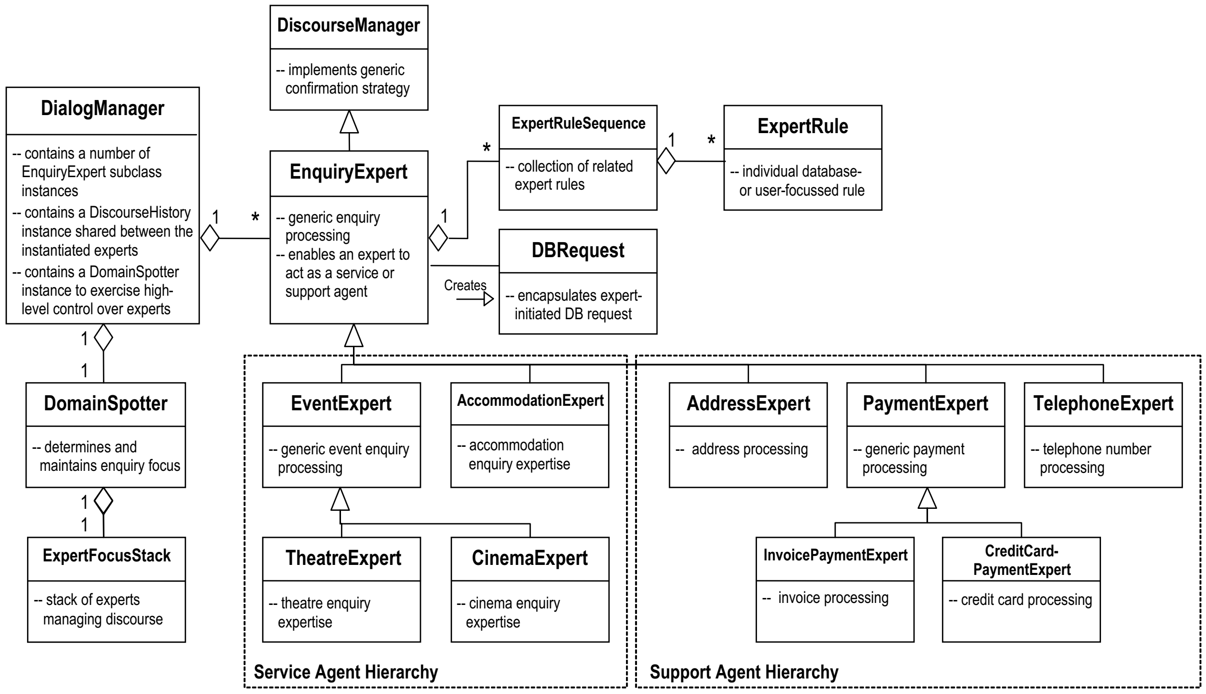
DiscourseManager
-- implements generic confirmation strategy
DialogManager
-- contains a number of EnquiryExpert subclass instances
-- contains a DiscourseHistory instance shared between the instantiated experts
-- contains a DomainSpotter instance to exercise high-level control over experts
EnquiryExpert
-- generic enquiry processing
-- enables an expert to act as a service or support agent
ExpertRuleSequence
-- collection of related expert rules
ExpertRule
-- individual database- or user-focussed rule
DBRequest
-- encapsulates expert-initiated DB request
Creates
1
*
DomainSpotter
-- determines and maintains enquiry focus
ExpertFocusStack
-- stack of experts managing discourse
EventExpert
-- generic event enquiry processing
AccommodationExpert
-- accommodation enquiry expertise
TheatreExpert
-- theatre enquiry expertise
CinemaExpert
-- cinema enquiry expertise
Service Agent Hierarchy
AddressExpert
-- address processing
PaymentExpert
-- generic payment processing
TelephoneExpert
-- telephone number processing
InvoicePaymentExpert
-- invoice processing
CreditCard-PaymentExpert
-- credit card processing
Support Agent Hierarchy

Fig. 1. The Expertise Hierarchy.

DomainSpotter that helps select domain expertise, and a DiscourseHistory that maintains a record of user-system interaction across domains.

The architecture is largely an inheritance hierarchy, as indicated by the open headed arrows of the UML: the class at the top of each arrow is a superclass, whose characteristics are inherited by the subclass at the bottom of the arrow. Thus, the DiscourseManager, at the top of the dialogue management hierarchy, determines the criteria used throughout the system for following the evolution of a dialogue: if the value of a particular attribute in the current frame is different from the corresponding attribute in the previous frame, then regard the value as having been changed by the user, and so on. Similarly the *DiscourseManager* superclass determines the style of the confirmation strategy used by all its immediate and subsequent subclasses, which include the different domain experts—like the Accommodation Expert or the TheatreExpert. Depending on the rules of the *DiscourseManager*, every DiscourseManager subclass will, therefore, echo back to the user any value that has been newly supplied or modified, and query any value that has been negated, for example. For the system developer, since generic behaviour is encapsulated at the top of the inheritance hierarchy, the process of adding domain-specific functionality is simplified: the developer can concentrate on adding the expert behaviour typical of the domain (e.g. what to ask for next, if the user supplies a date and a time; what to suggest, if the user wants to go to a movie on a day when the movie is not showing)—meanwhile the rules for following and responding to the evolution of the dialogue are provided 'for free' by inheritance. While the system's generic behaviour is domain-independent, each domain expert, though sharing implementational similarities with other domain experts, will have its own domain-specific expert rules: these represent the distinctive behaviours that distinguish a booking clerk in a theatre ticket office from a booking clerk at the front desk of a hotel.

The object-oriented approach assists the inclusion of new domains in another way. Domain experts that provide a front-line service—e.g. an

AccommodationExpert whose role is to facilitate hotel reservations—can avail themselves of 'support agents' that get particular jobs done regardless of the business domain: support agents might for example elicit address or credit card details.

The distinction between service and support can be used to maintain focus intuitively on a particular dialogue context. For instance, if the system is gathering payment details (the function of a support expert) for an accommodation booking (the function of a service expert), it may be preferable to complete the payment transaction before allowing the user to investigate other services. This and similar strategies are motivated by findings from an observational study of how staff in a busy tourist office imposed order on over-enthusiastic or hurried enquiries—for example, by saying things such as "Just let me finish taking your payment details...".

Service agents are not even concerned with the source of the support: if, for example, an AccommodationManager needs to elicit an address, it asks the Domain Spotter to find a support agent with the necessary expertise and the selected support agent handles that segment of the dialogue. Inheritance and 'uses' relationships facilitate the process of modifying the application, within the current application domains or by the addition of new domain expertise.

3.1.1. DiscourseManager

The DiscourseManager is responsible for the DM's overall 'conversational style', its generic discourse behaviour. 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. The DiscourseManager determines the system's response to new, modified or negated information from the user, and it determines when domain-specific rules of thumb, encapsulated in the suite of domain experts, should be allowed to fire. Since the system's confirmation strategy is often an implicit one, the system's utterances typically take the form of a confirmation of *new* or *modified* information supplied by the user (for example, "So, that's the Hilton in Belfast, checking-in date 19th October") followed immediately by the system's next question ("Could you tell me what checking-out date you would like?"). If the user has *negated* information that the system had previously recorded, the system's next utterance concentrates on correcting the negated information rather than seeking further information. Because the DiscourseManager is at the top of the inheritance hierarchy, its behaviour colours the manner in which its grandchildren (the EnquiryExpert subclasses such as the AccommodationExpert) interact with the user in their specialised business domains (accommodation booking, taking payment details, etc.).

Implicit confirmation of each attribute as it is supplied or modified is particularly reassuring when a user is encountering difficulty with the system's speech recognition or understanding. However, initial user trials also indicate that, when a dialogue is progressing well, the system's confirmatory repetitions of individual values, and of changes of domain, can be a source of frustration. Nonetheless, the approach is not dissimilar to that of a careful human telephone operator—especially if line quality is poor.

The DiscourseManager is able to work out what has been repeated, modified or negated by the user by comparing a frame of information relating to the user's latest utterance, with a corresponding frame representing the last discourse state. This comparison of frames is implemented as a set of evolve processes (see Section 3.2). The evolve rules govern the changes in confirmation status of a frame attribute value that occur as a result of the user's repetitions, confirmations, modifications and negations, and also changes in confirmation status that occur as a consequence of negations and modifications initiated by the system itself. If the user modifies more than one value, the system will simply echo back the corrected values and ask its next question: "Ok, got it. So that's the Europa hotel with a double room. Could you tell me what checking-in date you would like?" (assuming that the user had changed from a single room at the Ramada, for example).

The DiscourseManager makes use of a number of other components. In order to let each of its domain-specific EnquiryExpert grandchildren update the record of the evolving dialogue, it takes on the DialogManager's DiscourseHistory as an inheritable attribute of its own. The DiscourseHistory maintains a record of the evolving dialogue in the form of a stack of DialogFrames, each of which in turn comprises a set of Attribute objects relevant to the particular business domain. The EnquiryExpert and its subclasses represent the domain-specific expertise that augment the behaviour of the DiscourseManager once the latter's generic confirmation strategies have been applied.

3.1.2. Domain experts

The domain experts (or agents) encapsulate specialised behaviour, which can be readily extended by additional classes. There are two families of domain experts:

- 'service agents', such as the AccommodationExpert, that provide front-line services to the user, and
- 'support agents', such as the CreditCardExpert, that are able to elicit information required to complete one of the front-line service transactions.

The domain experts inherit generic dialogue handling skills from the DiscourseManager. Each of the domain Experts, whether a service agent or a support agent, has its own expert rules, contained in one or more expert rule sequences. Typically the expert rule sequences will be of one of two kinds:

- *user-focussed rules*: rules that are used to trigger the system's immediate response to specific confirmed combinations of information supplied by the user and recorded in the evolving dialogue frame—the rules may cause the system to ask for more information, or may initiate a database search, for example:

```
IF (the user has not given
    accommodation name [e.g. 'Hilton']
    and accommodation type [e.g. 'Hotel'])
THEN ask for accommodation type
                                                  (1)
```

- *database-focussed rules*: rules that are applied in the light of the system's failed attempts to retrieve information from or validate information against the database. These failed searches may result from a particular combination of search constraints, whether these are supplied by the user, or by the system when it attempts to retrieve information to assist the user. The source of the problem could also be a misrecognition by the system of the values supplied by the user. The system does not attempt to distinguish between different reasons for the application of the rules. Instead it simply relaxes constraints in the query as understood by the system in order to obtain some sort of match that might satisfy the user's requirements. An example of a relaxed query for a class of a particular 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–      (2)
    star and accommodation type hotel)
THEN relax constraint class four-star and re-do search
```

3.1.3. DiscourseHistory and dialogueFrame

Maintaining a record of the evolving discourse, and providing the means of creating and retrieving entries for individual user-system exchanges, are the responsibilities of the DiscourseHistory, which is a stack of DialogFrames. A DialogFrame is a set of attributes that corresponds to the frame of slots that must typically be filled to complete a transaction in a particular domain.

The frames consist of Attribute objects, each of which stores:

- the name (e.g. accommodation name, accommodation class, etc.) and elicited value (whether a string, integer, etc.) of a single piece of information (datum);
- the confirmation status of the datum (e.g. new_for_system);
- the level to which the datum has been confirmed (through repetition, or by the user's affirmative response to a system prompt—the level is represented by a simple numeric 'peg');
- the system intention regarding the datum (e.g. implicitly confirm new information; explicitly confirm information that has been negated; ask the user to specify information that is still required).

For example, the following DialogFrame

```
Attribute (Hilton, NEW_FOR_SYSTEM, 0,
  confirm)                                  (3)
```

indicates that the value 'Hilton' has the confirmation status of NEW_FOR_SYSTEM, with a discourse peg of 0, and a system intention to confirm the value.

Table 1 shows the set of confirmation statuses used in the current system, based on and extended from the set formulated by Heisterkamp and McGlashan (1996).

The discourse pegs reflect the degree of confirmedness of an attribute. For example, the discourse peg of an attribute 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 the current 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 re-confirmed, 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 be explicitly confirmed by the user for the system to treat that attribute as confirmed and having a discourse peg of 'greater than 0'. 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 along with the confirmation statuses are shown in Table 2.

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.

The Attribute objects thus give a multi-facetted view of each piece of information that it is being considered by the system. For example if an Attribute with the attributeName "Accommodation-Name" had the *systemIntention* CONFIRM in

Table 1
Confirmation statuses

NEW_FOR_SYSTEM	Initial user provided value for a given slot
INFERRED_BY_SYSTEM	Value deduced by system based on other populated slots
REPEATED_BY_USER	Existing value repeated/confirmed by user (either implicitly or explicitly)
MODIFIED_BY_USER	Value modified by user
NEGATED_BY_USER	Value negated by user
MODIFIED_BY_SYSTEM	Value modified by system (most likely to enable dialogue furtherance given an impasse arising from the previous value)
NEGATED_BY_SYSTEM	Value negated by system (most likely to enable dialogue furtherance given an impasse arising from the previous value)
CONFIRMED_BY_SYSTEM	Value confirmed by system (most likely via a successful database lookup)

Table 2
System intentions and associated confirmation statuses

CONFIRM	Implicitly confirm the stored value
EXPLICIT_CONFIRM:	Explicitly confirm the stored value with the user
REPAIR_CONFIRM	Confirm that the user has repaired a value
EXPLICIT_REPAIR_CONFIRM:	Explicitly confirm a repaired value with the user
REPAIR_REQUEST	Request that the user repair a negated value
REFORMULATION_REQUEST	Request that the user reformulate their enquiry (only used when the system is unable to proceed with the enquiry)
SPECIFY	Request that the user specify a particular value
SPECIFY_ALTERNATIVE	Request that the user specify a value other than the current value
UNSPECIFIED	No system intention

the last DialogFrame, and the user repeats the *attributeValue* (e.g. "Hilton Belfast"), then the DialogManager's rules for evolving a new DialogFrame will state that, in the corresponding Attribute of the new DialogFrame, the *discoursePeg* should be incremented, the *confirmationStatus* should be set to REPEATED_BY_USER and the system should reset its *systemIntention* for this Attribute to UNDEFINED (i.e. 'no further intention to confirm at this stage').

3.2. The use of the evolve process to determine confirmation strategies

The *evolve* process is the key process that determines the system's confirmation strategies. All the DialogFrames generated at each dialogue turn are stored in the DiscourseStack. When it receives new information, the system compares the information in the frame of the previous turn with the current input and produces a new frame—an evolved frame—using a suite of evolve rules. There are currently 12 basic evolve rules in the system. These rules are similar to the update rules in Information State Theory (Larsson and Traum, 2000). The previous frame contains the information provided by the user, the status of each piece of information (repeated, modified, negated), its level of confirmedness (its discourse peg), and the system's intentions for confirming or repairing supplied or missing values. This information enables the system to interpret the user's latest utterance in the light of values acquired so far, and accordingly to modify confirmation statuses and peg levels, and to formulate new intentions. The information in the previous frame also enables the system to interpret the user's latest utterance in the light of its *own* last intentions (e.g. if the user answers "Yes" in response to the system's utterance "So, you're staying in Dublin" (the result of the system's previous intention confirm the location), then Dublin can be regarded as the confirmed location). The latest attribute values, confirmation statuses, pegs, and intentions are entered in the new, evolved frame, which is then pushed on to the DiscourseStack. The system's next utterance is based on the content of the evolved frame.

Most of the *evolve* processes related to confirmation strategies involve examining the user's last value in the context of the system's last intention and evolving a new value and intention and updating the discourse peg. For example: the **Value to different value evolve process** applies when the user gives a value that is different from the value in the last DialogFrame, but does not negate that value. This situation could arise where the user changes his or her mind, or makes a correction of one of the items in the system's confirmation utterance that may have resulted from an earlier misrecognition. The following example, described in more detail in Section 4, illustrates this situation:

U1: I'd like to book a four-star hotel in Belfast from the 15th of December to the 20th of December.
S2a: Ok, let's start a new accommodation booking—that's a four-star hotel in Belfast from 15th December until 20th December. Please wait while I check the database…
S2b: You can choose between the following: the Europa, the Malone Lodge, the Stormont or the Holiday Inn. Which would you prefer?
U2: Could you make that a three-star? (4)

The **Value to different value evolve process** is applied in a number of different contexts (see Appendix A for the complete algorithm). In the case of example (4), the following context applies:

Both the old value and the new value were supplied by the user and the previous system intention was not SPECIFY, SPECIFY_ALTERNATIVE, or REFORMULATION_REQUEST. In this case the new value supplied by the user becomes the evolved value, the confirmation status is set to MODIFIED_BY_USER, the discourse peg is set to 0, and the new system intention is REPAIR_CONFIRM.

A series of other evolve processes cover a number of other contexts, of which the following are some examples:

1. The user says nothing in relation to the existing value so the value is considered to be implicitly confirmed. The resulting status is REPEATED_BY_USER, the discourse peg is incremented, and the intention blanked.
2. Following a system SPECIFY intention where no previous value existed, if the user supplies a new value the status of this new value is set to NEW_FOR_SYSTEM with a discourse peg of 0 and a new system intention to CONFIRM. Alternatively, if the value was supplied from the database, the confirmation status is set to INFERRED_BY_SYSTEM with a discourse peg of 0 and a new system intention of EXPLICIT_CONFIRM (or, in the case of more than one value, a system intention to SPECIFY).
3. This process applies if the new value is the same as the last value. There are three different contexts. If both values were supplied by the user, and if the last system intention was REFORMULATION_REQUEST or SPECIFY_ALTERNATIVE, then no changes are made, otherwise the value is retained and the discourse peg is incremented by 1 (indicating that the value had been repeated by the user and should thus have a higher level of confirmedness). Alteratively if the new value is supplied by the user and the last value was obtained from the database, the confirmation status is set to NEW_FOR_SYSTEM with a discourse peg of 0 and a system intention to CONFIRM. In the third context, where the new value is provided by the system and the last value was provided by the user, then the value is evolved as CONFIRMED_BY_SYSTEM with the discourse peg set to 1 and an evolved intention of UNSPECIFIED.

Thus depending on the context of a new dialogue turn in terms of the previous system intention, the confirmation status and the discourse peg, a new DialogFrame is evolved that specifies a new confirmation status, discourse peg and system intention for the evolved value. The confirmation status of an attribute together with the discourse pegs enable the system to decide on its next actions in 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. Algorithms such as the following are used to determine what to confirm and when.

```
IF
(the discourse pegs for AccoName,
Location, AccoRoomType, DateFrom
and DateTo >= 1)
 AND (Availability is confirmed by
the system)
 AND (Payment information has been
collected)
 THEN
 Generate a system utterance confirming
a reservation for AccoName,
Location, AccoRoomType, DateFrom and
DateTo
```
(5)

In this way the system makes reasoned decisions as to whether and how a value should be grounded based on the information provided in the evolved DialogFrame. An annotated example that illustrates this process is presented in Section 4.

4. The object-based dialogue manager in action

A number of scenarios were devised to test the logical flow of the system and to determine whether it behaved correctly given certain types of input. In particular, we were interested in observing how it responded to user corrections that indicated two different types of user intention:

1. Attempts by the user to correct a system interpretation, generally as displayed in a system confirmation.
2. Attempts by the user to change a value because the user has changed his or her mind.

Following Heisterkamp and McGlashan (1996) we do not attempt to model the user's intentions but simply capture the situation that the user wishes to change a value (for whatever reason). Our grounding mechanisms can operate independently of any model of what the user's intentions behind such a change might be. We also examined cases in which the system is unable to find information that matches the user's query and provides alternative suggestions through query relaxation. The following is an example of a test scenario, followed by an annotated excerpt from a typical user-system dialogue.

5. Scenario

A user wishes to book a four-star hotel in Belfast from 16–20 December. When the system returns a list of four-star hotels, the user changes the request to a three-star hotel. The system returns a list of three-star hotels and the user selects one and asks to book a double room. The user then asks for a single room rather than a double. As the system elicits payment details the user asks about movies. The system defers the movie enquiry, concludes the accommodation booking and then takes up the user's cinema enquiry, 'remembering' the cinema-related information that was supplied earlier.

In the following dialogue the user's input is followed by a representation of the recognised input and a brief commentary, indicating which main features of the DM are being called into play, is given at key points in the dialogue and is printed in italics.

S1: Welcome to the Queen's Communicator Demonstration System. How can I help you today?

U1: I'd like to book a four-star hotel in Belfast from the 15th of December to the 20 of December.
`Recognised Input = FOUR-STAR HOTEL BELFAST FROM 15TH DECEMBER TO 20TH DECEMBER.`

Key words like four-star and hotel help identify the accommodation domain. Accordingly the DomainSpotter appoints the Accommodation-Expert 'handling agent'. The AccommodationExpert populates its Accommodation DialogFrame with whatever relevant information has been supplied by the user. Information about the current state of the dialogue is encoded in a Dialogue Frame which lists the attributes of the frame along with their values, discourse pegs, confirmation statuses, and intentions (see Appendix B for an example of a Dialogue Frame following utterance U2).

S2a: Ok, lets start a new accommodation booking—that's a four-star hotel in Belfast from 15th December until 20th December. Please wait while I check the database…

The AccommodationExpert, using the generic confirmation strategies that it has inherited from the DiscourseManager, implicitly confirms the new values that it has identified, and uses its own domain-specific rules to initiate a database lookup that will retrieve names of hotels that match the user's criteria.

S2b: You can choose between the following: the Europa, the Malone Lodge, the Stormont or the Holiday Inn. Which would you prefer?

U2: Could you make that a three-star?
`Recognised Input = THREE-STAR`

Although the system in its utterance has strongly suggested that the user should select one of the hotel names it has retrieved from the database, this is a mixed initiative system, so the user may provide information different to that which was requested. In this case the user changes the star-rating required. As a result of the user changing the original value, the confirmation status of the attribute AccoClass changes to MODIFIED_BY_USER with a new system intention for this attribute of REPAIR_CONFIRM. Changes are also made to other attributes, such as AccoType, Location, DateFrom, and DateTo: the pegs have been incremented and the status changed to REPEATED_BY_USER. This status is assigned on the basis that these values are now considered to have been implicitly confirmed by the user since they were not modified when the user modified the value AccoClass (see Appendix B).

S3: OK, got it. So that's a three-star. Is that correct?

Using its inherited confirmation strategy, the AccommodationExpert compares the value for the star-rating in the previous frame with the star-rating in the current frame and performs a 'repair-confirm' on the changed value.

U3: Yes, that's right.

`Recognised Input = YES`

As a result of the user's confirmation of the three-star rating, the discourse peg of the attribute AccoClass is incremented and its confirmation status changed to REPEATED_BY_USER, i.e. explicitly confirmed.

The remaining dialogue segments illustrate domain switching between two service domains and the interplay between service and support agents.

S4a: Please wait while I check the database.
S4b: You can choose between the following: the Jury's or the Lisdara. Which would you prefer?
U4: I'll go for a double room at the Jury's.
S5: So that's the three-star Jury's hotel in Belfast with a double room, from 15th December until 20th December. Is that correct?

In the current system implementation the handling agent will explicitly confirm all key values—i.e. those needed to conclude a major segment of the discourse—once these have been supplied by the user.

U5: Make that a single room.

Again however, the user changes his/her mind. Rather than perform a 'repair-confirm' on the individual changed value and re-confirm it in the general confirmation of key values, generic EnquiryExpert behaviour determines that the handling agent should perform only the general confirmation.

S6: So that's the three-star Jury's hotel in Belfast with a single room, from 15th December until 20th December. Is that correct?
U6: Yes.
S7: Would you like me to check availability?
U7: Yes.
S8a: OK, I'll see if that's available.
S8b: Yes that's available. Do you wish to make a booking?
U8: Yes, please.
S9a: OK, I'll see if I can reserve that.
S9b: Yes, that's reserved. Ok, let's start collecting your address details. Could you tell me your house number please?
U9: It's 3 Malone Road, Belfast BT7 1NN.

Although the system asked only for the house number, the user provided his/her full address. In a frame-based, phrase-spotting system such as this, any relevant information will be used by the handling agent (at this point an AddressExpert) to populate slots in the current dialogue frame. The unrequested information is thus accepted...

S10: So, that's three Malone Road, Belfast; BT7 1NN. Is that correct?
U10: Yes.
S11: OK, I've got your address details. Ok, let's start collecting your payment details. How would you like to pay?

U11: Could I just check what's on at the UGC Cinema?

Again the user takes the initiative by ignoring the request from the handling agent—which is now the PaymentExpert—for details of the type of payment (e.g. credit card or invoice). Instead the user asks about an entirely different transactional domain, namely cinema bookings. On the basis of key words in the user's utterance ('UGC', 'Cinema', which will result in the parse segment '[Event_venue].[Cinema].ugc'), the DomainSpotter recognises that the user is attempting to change topic from a support agent to a new service agent, and at this stage defers the request. However, the DomainSpotter places the CinemaExpert at the bottom of the ExpertFocusStack for use later, and thanks to the system's UtteranceStore, the CinemaExpert, when it eventually becomes handling agent, will be able to make use of the information that the user has already supplied.

S12: Do you want to leave your payment details and change to cinema booking? Have I correctly understood you?

U12: Yes.

S13: Ok, I got you. We will be able to proceed to cinema booking shortly. At the moment I need to finish your current payment details. Ok, let's start collecting your payment details. Could you tell me your preferred method of payment?

Once the payment details for the accommodation have been elicited and confirmed, and the accommodation booked, the system is ready to move on from the AccommodationExpert to the next agent on the ExpertFocusStack, in this case the Cinema- Expert—which, like all agents, first of all checks the UtteranceStore for any relevant, previously parsed input.

S14: Ok your accommodation booking is complete. Ok, let's start a new cinema booking. Ok, that's in the UGC cinema. Could you tell me what show date you would like?

U14: The 17th of December.

S15a: So that's 17th December. Please wait while I check the database.

S15b: You can choose between the following shows: Brother Bear, Love Actually, Elf, SWAT, The Fighting Temptations, The Mother, Thirteen, Timeline, Lord of the Rings 3. Which would you prefer?

6. Evaluation

In order to evaluate the performance of the system, and of the Dialogue Manager in particular, two scenarios were devised. The first scenario was a set transaction in which the subject had to perform a number of tasks, as follows:

You want to book a hotel. You're staying in Belfast. The hotel should be three-star or four-star. You'd like to stay from 2nd September to 10th September. You need a double room. Give details of who you are. You can be called John Spence or Victoria Jones. You live at a house numbered between 1 and 20 on either Market Street or University Street in Belfast. You can pay by Visa or Mastercard. You can give any eight-digit credit card number, any four-digit credit card expiry date, and any eight-digit telephone number (write these numbers down in advance so that you can say them fluently). You also want to go to see Starlight Express during the period of your stay.

The second scenario involved a more open-ended set of instructions:

Book a {movie or theatre-show} and accommodation in Belfast. Provide information that you think would be helpful, or use the information that the system provides.

The main objective of the evaluation was to assess the system's error handling capabilities, in particular, whether users could correct values that had been incorrectly recognised and whether they could switch easily between the accommodation and event domains. A total of 24 dialogues were conducted, involving 12 subjects. All transactions, except for one, in which the user abandoned the transaction, were completed successfully. In general users were able to correct values that had been incorrectly recognised, and were able to get the system to repeat utterances that they had been unable to understand. They were also able to change easily between the accommodation and event domains.

The system's behaviour was evaluated against the Trindi 'theoretical perspective' checklist (Bohlin et al., 1999) and performed well in the following respects:

1. The system interprets user utterances, according to context.
2. The system is capable of dealing with more/different/less information than was requested.
3. The system deals with non-help subdialogues initiated by the user.
4. The system asks appropriate follow-on questions.
5. The system can deal with ambiguous or inconsistent information. If, for instance the user provides information that may be relevant to more than one area of expertise, the system will use its DomainSpotter to offer the user more specific alternatives from which to choose, e.g.

User: What events are on?
System: Ok, let's start a new event booking. Do you wish to consider cinema booking or theatre booking. And again, if the user provides information that is inconsistent (e.g. wanting to go to the theatre on a day when no shows are playing), the system will attempt to use its domain-specific, database-focussed rules to relax a constraint and offer appropriate alternatives, e.g.
User: I'd like to go to the Grand Opera House on the 12th of September.
System: Sorry that's not available. You can choose between the following show dates...

The system's strategy of parsing key words and phrases from the user's input means that it copes relatively well with inaccurately recognised phrases or noisy input. However, significant omissions from the Trindi theoretical perspective checklist include help subdialogues (the user cannot, for instance, ask 'What can I do now?') and negatively specified information (the user cannot, for instance, currently state 'I'll go any day other than Monday').

User satisfaction achieved an average rating of 3.5 out of 5, based on a series of questions asked of the users after each session (e.g. "Were you able to complete the accommodation task to your satisfaction?").

Analysis of differences between users showed that some users were able to complete the task rather more easily, in a markedly shorter time, and in fewer turns than others. Key factors that appear to have delayed dialogue completion were inaccurate speech recognition, the system's limited language understanding, and users' inexperience of using the system.

The speech recognition problem was not altogether unexpected, since a general purpose speaker dependent recogniser was used, for which each user conducted a basic 10 min training session. The user's stored speech profile was selected when they logged on to the system. Recognition accuracy varied considerably across users. User ION had a word error rate typically around 3%, and had used the system several times before, so it was not surprising to see that system failures to understand were few, and that tasks were completed quickly (see Fig. 2). On the other hand, user JMC had a very high word error rate of around 37%, and was in addition an inexperienced user. Again, there were many more failures to understand, and many more turns were required to complete the dialogue.

However, a high word error rate does not lead necessarily to difficulties at the dialogue level. The system's semantic parser, which associates meanings with key words and phrases, was able to progress the dialogue as long as key words and phrases are recognised, providing some degree of robustness against misrecognition. In instances where the system was unable to extract any useful information from the user's utterance (i.e. the utterance could not be parsed, or, if parsable elements were extracted, the utterance still could

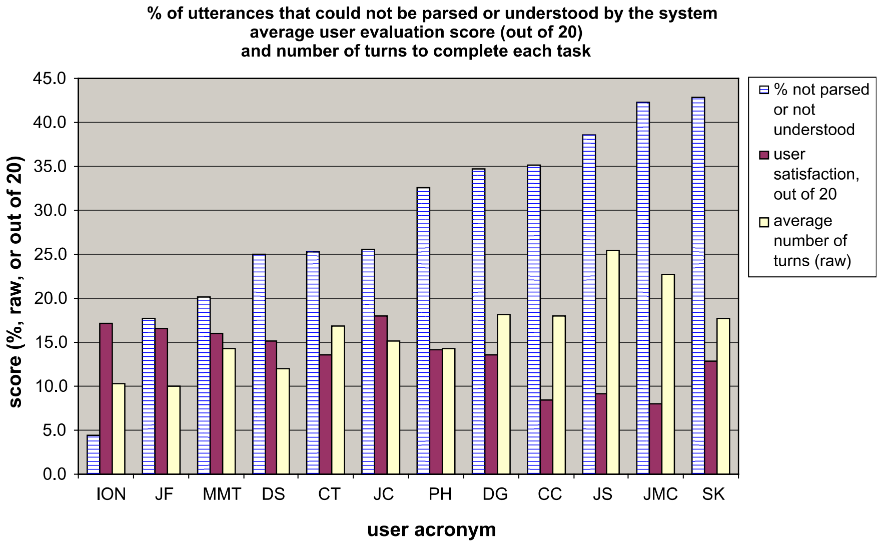

Fig. 2. System performance measures.

not be understood or used meaningfully in the current context), then the system asked the user to repeat their utterance. While the system's vocabulary was adequate for testing, users occasionally struggled to find a word or phrase that the system could both recognise and understand. Naturally enough, failures to recognise and/or understand correlated broadly with more dialogue turns needed to complete the task, and a consequent reduction in user satisfaction, as shown in Fig. 2. Improvements in performance would be expected by expanding the system's working vocabulary in subsequent releases, and by training a speaker-independent and domain-specialised recogniser.

Regarding confirmation strategies, users generally found helpful the system's generic strategy of echoing back for implicit confirmation information that the user had changed or had newly supplied, as well as its strategy of using brief scene-setting phrases, such as "Ok, let's start a new accommodation booking", to ground the dialogue in the current context. However, especially when the recogniser was performing well, more competent users reported that they found the system's solicitousness irritating. A more flexible, adaptive (generic) dialogue strategy would help to overcome this problem.

As shown in Table 3 the system was in general more talkative than the user—in part on account

Table 3
System performance scores

Task	Average duration	Average turns	Average system words	Average user words	Average system words/turn	Average user words/turn
Accommodation	00:03:00	10.1	207.3	64.8	21.5	6.4
Theatre	00:04:44	16.3	302.4	82.7	19.0	4.8
Cinema	00:05:01	17.3	325.7	85.5	19.3	4.9
Personal details (full)	00:05:19	20.1	321.1	122.6	16.3	6.1
Personal details (short)	00:01:11	3.7	89.8	10.6	26.0	2.7

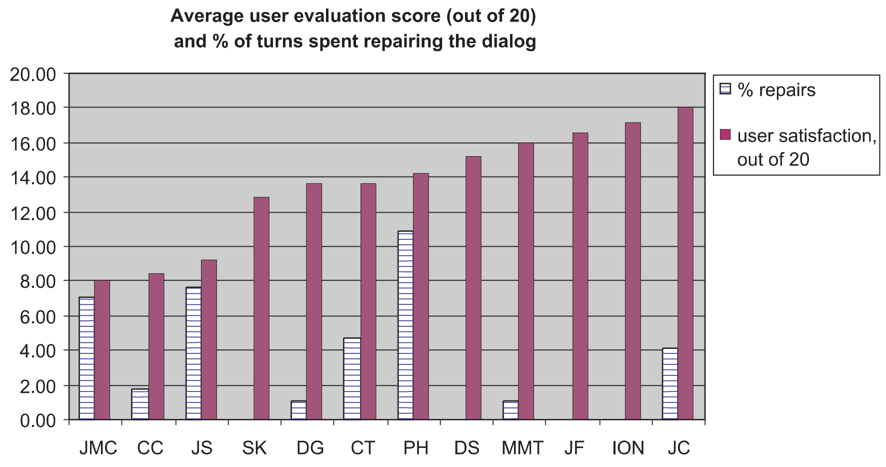

Fig. 3. User evaluation and repair turns.

of the system's confirmation and grounding strategies, in part also because inexperienced users tended to follow the system's prompts quite closely, rather than over-specify their requirements, as the system indeed allows.

Being asked to repeat information (as a result of turns in which the system could not parse or understand anything) seems to have been a more consistent source of frustration to the user than simply having to correct the system, as a result of a misunderstanding, or as a result of the user changing his or her mind.. Indeed, hearing the system respond to a changed value ("OK, got it, so that's a double room..."), or remark on a negation ("Oh, so that's not a double room. I'll see what I can suggest..."), reassured at least one user that the system was doing its best to understand and help. Fig. 3 shows user satisfaction levels compared with the percentage of turns spent repairing the dialogue: there is no obvious correlation. Repairs include the user modifying information that they had previously provided—or information that the system had understood as having been provided—and the user negating previously supplied or understood values.

It also transpired that the theatre booking and cinema tasks were rather more difficult to complete than the accommodation bookings (see the duration figures in Table 3). Contributing to this difference are the following factors:

- The cinema and theatre tasks required that the user supply information (like a required date and theatre) and then listen to the information that the system would draw from its database (like the show that was playing). If the show did not suit, or if nothing was playing at the venue on the required date, etc., the user had to follow the system suggestion, or try an alternative of their own. The user did not know in advance the schedule that the system was using to provide the information.
- By contrast, the accommodation booking task had very few database restrictions, as the hotels always had free rooms during the period required and the nature of the task was obvious, such as booking a double room with specified dates.

It can also be seen from Table 3 that there were long and short forms of the 'personal details' task. Having given full details of the accommodation that they required, users had to complete the booking by giving personal details such as name and address, means of payment, etc. The long form of the task consisted of giving all the details required, as would occur on using the system for the first time in a session. For the short form of the task the system asked the user if they wished to use previous details (“same name and address as before?”, “same payment details as before?”) and the user could simply answer “yes” to both questions.

One interesting issue that has emerged from the evaluation concerns the system’s model of discourse evolution, particularly in cases where repetition of a value by the user equates with explicit confirmation of the value. For example, if the user repeats a value preceded by “not”, as in “Not the Lyric Theatre”, in order to indicate that the system has understood incorrectly, and the system fails to recognise the “not”, then the discourse can take a turn which the user had explicitly not intended. Since the system is now confident that the value that the user assumes negated has just been confirmed by repetition, it will not immediately confirm the value further, with the result that this talking at cross-purposes may only come to light several turns later, when it might be awkward to correct!

7. The advantages of an object-based approach to grounding

In general the attraction of object-orientation for dialogue management is that it can be used to separate generic dialogue behaviour from domain-specific behaviour. In an OO implementation, routine functionality such as turn-taking and confirming what the user has said is made available through inheritance to more specialised dialogue components. Maintainability is therefore enhanced: generic behaviour need be implemented only once; new agents can inherit the generic dialogue behaviour and possibly refine the specialised behaviour of existing experts: for example, TheatreExpert can be introduced as a subclass of EventExpert. The more specialised components encapsulate ‘expert rules’ for enquiry processing in a particular business domain (e.g. “if a location and a hotel have been requested, then check what class of hotel is required”) or rules that represent know-how that can be used in a number of business domains (e.g. how to elicit credit card details). In order to enable mixed initiative interactions across domains, the system’s behaviour is modelled as a collaboration between a cohort of ‘agents’, each an instance of a Java class. An agent is a specialist in a particular transactional area—e.g. booking accommodation or eliciting an address—and uses its own domain-specific expert rules to elicit information (e.g. information for making a hotel booking) that is stored in a specialised dialogue frame. Thus each agent encapsulates a skillset for a substantial dialogue or subdialogue.

Like the Communicator team at Carnegie Mellon University (Rudnicky and Xu, 1999) we view the dialogue product, which represents the knowledge to be elicited as well as the current state of the dialogue, including the dialogue history, as a tree-like structure. However, in the Queen’s Communicator the nodes of the dialogue tree are complete dialogue frames, each of which is implemented as a specialisation of a generic DialogFrame (specialised dialogue frames include TheatreDialogFrame, and CreditCardDialogFrame). A key feature of the DM in the Queen’s Communicator is that the discourse structure and the corresponding dialogue product evolve dynamically as agents are selected by a DomainSpotter, in the light of the user’s utterances or as a consequence of the agents’ own rules. It is this process rather than an overarching dialogue plan or agenda that drives the discourse forward, sometimes across domain boundaries.

The system’s grounding strategies make use of this representation in order to determine what items to confirm and how to confirm them. The DialogFrames encode information about the current state of the dialogue, such as what the user said, the confirmation status of the values that have been elicited, the degree of confirmedness of each value, and the associated system intentions that will be used to generate the system’s actions and utterances. The object-oriented architecture allows generic confirmation strategies to be inherited from the DiscourseManager and specific strategies to be applied as required by specialist agents, such as EventExpert or Accommodation Expert.

The same architecture also supports other functionalities of the system, such as domain spotting and dealing with user-led focus shifts (see O'Neill et al., 2005).

8. Future work

There are two new directions that we would like to investigate in relation to the use of grounding strategies in the Queen's Communicator. 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 recognised 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 to control the progression of a dialogue and to make decisions regarding the treatment of miscommunication (Paek and Horvitz, 2000).

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 structures of the DialogFrames dynamically 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 (Larsson and Traum, 2000) except that we would also retain the benefits of our object-based architecture.

Appendix A. The Value to Different Value evolve process

```
  if (lastValue != inputValue && inputValue
  NOT negate lastValue) (i.e. Value → Value2)
  {
    if (both are userValue)
      if (lastIntention == SPECIFY, SPECIFY_
      ALTERNATIVE or REFORMULATION_
      REQUEST)
    // If we have a SPECIFY, SPECIFY_
    ALTERNATIVE or REFORMULATION_
    REQUEST
    // then we have a NFS status, and zero peg.
       evolvedValue ← inputValue
       evolvedStatus ← NEW_FOR_SYSTEM
       evolvedPeg ← 0
       evolvedIntention ← CONFIRM
         else //All others will be evolved as a
         MBU, and 0 peg
         evolvedValue ← inputValue
     evolvedStatus ← MODIFIED_BY_USER
       evolvedPeg ← 0
       evolvedIntention ← REPAIR_CONFIRM
         else if (lastValue is DBValue and input-
  Value is userValue)
// If the user has specified a value in response to a
//system suggested value then this should be
evolved as NFS
         evolvedValue ← inputValue
         evolvedStatus ← NEW_FOR_
         SYSTEM
         evolvedPeg ← 0
         evolvedIntention ← CONFIRM
      else if (inputValue is DBValue)
// If the input value is system suggested then evolve to
```

Appendix B. Dialogue Frame following utterance U2

```
DiscourseManager->evolveInputFrame:Evolved frame with Intentions:
F_Name:Accommodation,    F_ID: 5, ExprtID: AccommodationExpert
```

AccoName = []	Peg = 0	Status = UNS	Intention = UNSP
AccoType = [HOTEL]	Peg = 1	Status = RBU	Intention = UNSP
AccoClass = [three-star]	Peg = 0	Status = MBU	Intention = RCNF
Location = [BELFAST]	Peg = 1	Status = RBU	Intention = UNSP
AccoRoomType = []	Peg = 0	Status = UNS	Intention = UNSP
DateFrom = [2003-12-15]	Peg = 1	Status = RBU	Intention = UNSP
DateTo = [2003-12-20]	Peg = 1	Status = RBU	Intention = UNSP
CheckAvailability = []	Peg = 0	Status = UNS	Intention = UNSP
Reservation = []	Peg = 0	Status = UNS	Intention = UNSP
Completion = []	Peg = 0	Status = UNS	Intention = UNSP
PaymentDetails = [-1] [Payment]	Peg = 0	Status = UNS	Intention = UNSP

```
// either a MBS or NBS depending upon the number
of suggestions
        if (only one DBValue)
          evolvedValue ← inputValue
          evolvedStatus ← MODIFIED_BY_SYSTEM
          evolvedPeg ← 0
          evolvedIntention ← EXPLICIT_CONFIRM
          else if (more than one DBValue)
          evolvedValue ← inputValue
          evolvedStatus ← NEGATED_BY_SYSTEM
          evolvedPeg ← −1
          evolvedIntention ← SPECIFY_ALTERNATIVE
}// end of this process
```

References

Bohlin, P., Bos, J., Larsson, S., Lewin, I., Matheson, C., Milward, D., 1999. Survey of existing interactive systems. Task-Oriented Instructional Dialogue, TRINDI Technical Report, LE4-8314.

Bouwman, A., Sturm, J., Boves, L., 1999. Incorporating confidence measures in the Dutch train timetable information system developed in the Arise project. In: Proc. Internat. Conf. Acous., Speech, and Signal Process. (ICASSP), Vol. 1, Phoenix, AZ., pp. 493–496.

Clark, H.H., 1996. Using Language. Cambridge University Press, Cambridge.

Evermann, G., Woodland, P.C., 2000. Large vocabulary decoding and confidence estimation using word posterior probabilities. In: Proc. ICASSP 2000, pp. 1655–1658.

Hazen, T.J., Burianek, T., Polifroni, J., Seneff, S., 2000. Integrating recognition confidence scoring with language understanding and dialogue modeling. In: Proc. 6th Internat. Conf. on Spoken Language Process., Beijing, China.

Heisterkamp, P., McGlashan, S., 1996. Units of dialogue management: An example, ICSLP96. In: Proc. 4th Internat. Conf. on Spoken Language Processing, Philadephia, pp. 200–203.

Komatani, K., Kawahara, T., 2000. Flexible mixed-initiative dialogue management using concept-level confidence measures of speech recognizer output. In: Proc. COLING, pp. 467–473.

Krahmer, E., Swerts, M., Theune, T., Weegels, M.E., 2001. Error detection in spoken human–machine interaction. Internat. J Speech Technol. 4 (1), 19–30.

Larsson, S., Traum, D.R., 2000. Information state and dialogue management in the TRINDI Dialogue Move Engine Toolkit. Nat. Language Eng. 6 (3-4), 323–340.

Litman, D.J., Hirschberg, J., Swerts, M., 2000. Predicting automatic speech recognition performance using Prosodic Cues. In: Proc. 1st Meet. of the North Amer. Chap. of the Assoc. for Comput. Linguistics, pp. 218–225.

Litman, D.J., Walker, M.A., Kearns, M.S., 1999. Automatic Detection of Poor Speech Recognition at the Dialogue Level. Proc. 37th Ann. Meet. of the Assoc. for Comput. Linguistics (ACL'99), College Park, MD, June, pp. 309–316.

McRoy, S.W., 1998. Preface-detecting, repairing and preventing human–machine miscommunication. Int. J. Human–Comput. Studies 48, 547–552.

O'Neill, I.M., McTear, M.F., 2000. Object-oriented modelling of spoken language dialogue systems. Nat. Language Eng. 6 (3–4), 341–362.

O'Neill, I.M., McTear, M.F., 2002. A pragmatic confirmation mechanism for an object-based spoken dialogue manager. In: Proc. of ICSLP-2002, Vol. 3, Denver, September, pp. 2045–2048.

O'Neill, I.M., Hanna, P., Liu, X., McTear, M.F., 2003. The Queen's communicator: An object-oriented dialogue manager. In: Proc. of EuroSpeech2003. Geneva, September.

O'Neill, I.M., Hanna, P., Liu, X., Greer, D., McTear, M.F., 2005. Implementing advanced spoken dialogue management in Java. Science of Computer Programming 54 (1), 99–124.

Paek, T., Horvitz, E., 2000. Conversation as action under uncertainty. In: Proc. 16th Conf. on Uncertainty in Artificial Intelligence, Morgan Kaufmann, pp. 445–464.

Pellom, B., Ward, W., Pradham, S., 2000. The CU communicator: An architecture for dialogue systems. In: Proc. of the 6th Internat. Conf. of Spoken Language Processing, Beijing, China.

Rudnicky, A., Xu, W., 1999. An agenda-based dialog management architecture for spoken language systems. In: Proc. of IEEE Automatic Speech Recognition and Understanding Workshop.

Skantze, G., 2003. Exploring human error handling strategies: implications for spoken dialogue systems. In: Proc. of ISCA Workshop on Error Handling in Spoken Dialogue Systems, Chateau d'Oex, Switzerland, pp. 71–76.

Sturm, J., den Os, E., Boves, L., 1999. Dialogue management in the Dutch ARISE train timetable information system. In: Proc. of 6th Internat. Conf. on Speech Comm. and Technol. (99), Budapest, Hungary, pp. 1419–1422.

Swerts, M., Litman, D.J., Hirschberg, J., 2000. Corrections in spoken dialogue systems. In: Proc. of the 6th Internat. Conf. of Spoken Language Processing, Beijing, China.

Traum, D.R., 1999. Computational Models of Grounding in Collaborative Systems. In: AAAI Fall Symp. on Psychol. Models of Comm., 124–131, November.

Turunen, M., Hakulinen, J., 2001 Agent-based error handling in spoken dialogue systems. In: Proc. 7th Internat. Conf. on Speech Comm. and Technol. (Eurospeech2001), Aalborg, Denmark, pp. 2189–2192.

Walker, M.A., Wright, J., Langkilde, I., 2000a. Using Natural Language Processing and Discourse Features to Identify Understanding Errors in a Spoken Dialogue System. In: Proc. 17th Internat. Conf. on Machine Learning.

Walker, M.A., Langkilde, I., Wright, J., Gorin, A., Litman, D.J., 2000b. Learning to Predict Problematic Situations in a Spoken Dialogue System: Experiments with How May I Help You? North American Meeting of the Association of Computational Linguistics.

Ward, W., 1994. Extracting information from spontaneous speech. In: Proc. of ICSLP-94, Yokohama (Japan), September.

Wessel, F., Macherey, K., Schluter, R., 1998. Using word probabilities as confidence measures. In: Proc. ICASSP'98, pp. 225–228.