INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 666-2900 • FAX (510) 666-2956

Spoken Language Processing in the Clarissa Procedure Browser

Manny Rayner
Beth Ann Hockey
Jean-Michel Renders
Nikos Chatzichrisafis
Kim Farrell*

TR-05-005

April 2005

Abstract

Clarissa, an experimental voice enabled procedure browser that has recently been deployed on the International Space Station, is as far as we know the first spoken dialog system in space. We describe the objectives of the Clarissa project and the system's architecture. In particular, we focus on four key problems: creating voice-navigable versions of formal procedure documents; grammar-based speech recognition using the Regulus toolkit; methods for accurate identification of cross-talk based on Support Vector Machines; and robust side-effect free dialogue management for handling undos, corrections and confirmations.

*Affiliations and email addresses as follows. Manny Rayner: ICSI, mrayner@icsi.berkeley.edu. Beth Ann Hockey: UCSC, bahockey@email.arc.nasa.gov. Jean-Michel Renders: Xerox Research Center Europe, Jean-Michel.Renders@xrce.xerox.com. Nikos Chatzichrisafis: formerly RIACS, Nikos.Chatzichrisafis@web.de. Kim Farrell: RIACS, kfarrell@email.arc.nasa.gov. Work at ICSI, UCSC and RIACS was supported by NASA Ames Research Center internal funding. Work at XRCE was partly supported by the IST Programme of the European Community, under the PASCAL Network of Excellence, IST-2002-506778. This publication only reflects the authors' views.

1 Introduction

On the International Space Station (ISS), astronauts execute thousands of complex procedures to maintain life support systems, check out space suits, conduct science experiments and perform medical exams, among their many tasks. Today, when carrying out these procedures, an astronaut usually reads from a PDF viewer on a laptop computer, which requires them to shift attention from the task to scroll pages.

This report describes Clarissa, an experimental voice-enabled procedure reader developed at NASA Ames during a three year project starting in early 2002, which enables astronauts to navigate complex procedures using only spoken input and output (there is also a conventional GUI-style display, for situations where it is feasible for the astronaut to watch the screen). We hope that this will offer significant advantages for carrying out hands- and eyes-busy tasks in the challenging environment of microgravity. Clarissa has a vocabulary of about 260 words and supports about 75 different commands, including reading steps, scrolling forwards or backwards in the procedure, moving to an arbitrary new step, reviewing non-current steps, adding or removing voice notes, displaying pictures and setting alarms or timers. The software was installed on the ISS in January 2005. To the best of our knowledge, Clarissa is the first spoken dialogue system in space.

The overall architecture is described in Section 2; first, however, we present a sample session, chosen to illustrate the key design issues. The system runs on a standard A31p IBM laptop under Windows 2000[1]. The astronaut user puts on a headset including a noise-cancelling microphone, and starts Clarissa by clicking on an icon. After the system has come up, all user input is in spoken form; the system responds both by speaking and by changing the visual display. In the dialogue shown in Figures 1, 2 and 3, user turns are marked as $\mathbf{U}_n$, system turns as $\mathbf{S}_n$.

$\mathbf{U}_1$	Load EMU checkout procedure
$\mathbf{S}_1$	Loading EMU checkout procedure...
$\mathbf{U}_2$	Stop talking
$\mathbf{S}_2$	<beep>
$\mathbf{U}_3$	*[to colleague]* The EMU Checkout procedure is for maintenance on the American space suits. It can be done on from 1 to 3 of them at a time.
$\mathbf{S}_3$	*[system does not respond]*
$\mathbf{U}_4$	Go to step 14
$\mathbf{S}_4$	You need to complete step 1 first, step 15 ...
$\mathbf{U}_5$	Undo
$\mathbf{S}_5$	We are at note before step 1. Step 1: are you working on EMU1?
$\mathbf{U}_6$	Yes
$\mathbf{S}_6$	Answer was “yes”. Step 1: on EMU2?
$\mathbf{U}_7$	No
$\mathbf{S}_7$	Answer was “no”. Step 1: on EMU3?
$\mathbf{U}_8$	No
$\mathbf{S}_8$	Answer was “no”. Powering up ...

Figure 1: Example session with Clarissa (part 1)

[1]This is the only configuration permitted on the ISS.

$\mathbf{U}_9$ Go to step 48
$\mathbf{S}_9$ Note before step 48
$\mathbf{U}_{10}$ Next
$\mathbf{S}_{10}$ Step 48: While depressing SOP manual override (50 seconds maximum).
At SOP, check SOP interstage gauge. At DCM, check status suit p, nominal and stable.
To minimize SOP usage, remember the values. I will ask for them once you release the manual override and say complete.
$\mathbf{U}_{11}$ Complete
$\mathbf{S}_{11}$ No completion record in normal mode. Step 48: what is the suit pressure for EMU1?
$\mathbf{U}_{12}$ 5.2
$\mathbf{S}_{12}$ 5.2, value is off nominal high. Report value to MCC before proceding to next step.
$\mathbf{U}_{13}$ No, I said 3.7
$\mathbf{S}_{13}$ *[changes the display to 3.7]* 3.7, value is nominal.
Step 48: What is the interstage gauge reading for EMU1?
$\mathbf{U}_{14}$ 500
$\mathbf{S}_{14}$ Is dry LCVG to be filled for EMU1?
$\mathbf{U}_{15}$ Yes
$\mathbf{S}_{15}$ Step 49: for EMU1 only: cooling loop jumper, disconnect from multiple water connector.
$\mathbf{U}_{16}$ Next
$\mathbf{S}_{16}$ Step 50: for EMU1 only, dry LCVG, connect to multiple water connector

Figure 2: Example session with Clarissa (part 2)

This dialogue illustrates many of the key design issues in the Clarissa project. First, consider the relationship between the written and spoken versions of the procedure, which often differ in various ways. For example, the written version states at the beginning that the procedure can be valid for one to three suits, and trusts the users' (considerable) intelligence to know how to interpret the text accordingly. The spoken version must in contrast be explicit about adjusting the spoken output to the specific suits used, as for example in turns 11 and 14 and 15. There are also several written constructions which cannot be directly transposed into the spoken modality, like the table in turns 10–14 and the steps bracketed by the vertical bar in turns 15–17. In addition, ISS procedures are complex formal documents, typically representing hundreds or even thousands of hours of work, which have to undergo a lengthy sign-off process before they can be approved for use: this raises the question of how adequately to demonstrate that the spoken version is still valid, despite the fact that it differs in various ways from the written one. All these issues are discussed further in Section 3.

The next point concerns the user language supported by the system. In general, any type of spoken dialogue system has to steer a course between two basic strategies. At one end, the system can try to adapt to the user. The implementors collect data representing what users would ideally like to be able to say; a common way to do this is to use Wizard of Oz methods. The system is then configured to perform as well as possible on the sample data, usually using statistical techniques. At the other end of the scale, the burden of adaptation is put on the user: the system is designed to offer a predefined range of coverage, which the user is required to learn. In practice, of course, some compromise between these two positions is normal.

For many applications, the robustness inherent in the data-driven approach makes it the prefer-

$\mathbf{U}_{17}$	Go on
$\mathbf{S}_{17}$	Step 51: for EMU1 only, perform 1.525, LCVG WATER FILL, steps 6 to 17. Do you want to load this procedure?
$\mathbf{U}_{18}$	Yes
$\mathbf{S}_{18}$	Starting procedure: LCVG water fill procedure. Step 6 At DCM...
$\mathbf{U}_{19}$	Go to step 20
$\mathbf{S}_{19}$	Step 20: this is after the last step you are suppose to execute. Say close the procedure to exit the procedure, otherwise LCVG, disconnect from...
$\mathbf{U}_{20}$	Close procedure
$\mathbf{S}_{20}$	Do you want to close LCVG water fill procedure?
$\mathbf{U}_{21}$	Yes
$\mathbf{S}_{21}$	Closing current procedure. The EMU checkout procedure is now active.

Figure 3: Example session with Clarissa (part 3)

able choice. Astronauts, however, are very far from being typical users; they are highly intelligent, willing to invest time in learning to use the system, and in most cases come from an aeronautics background where use of controlled language is the norm. For this kind of user, a designed coverage approach has clear attractions; performance for expert users is better, and it is easier to reconfigure the system in response to changes in the specification. These issues are discussed in Section 4.

Robustness with respect to variation in user language is not particularly important for an application like Clarissa. However, two other kinds of robustness are critical. Since the whole point of the system is to support hands- and eyes-free operation, recognition has to be performed in an "open-mic" mode. This implies that the system needs to be able to reject spoken input ("cross-talk") not intended for it, as illustrated in turn 3. Our approach to handling cross-talk is described in Section 5. Robustness is also crucial at the level of dialogue management. For the usual reasons, recognition can never be completely accurate; if the system misrecognises, it must be easy for the user to undo or correct the misrecogition, as for example in turns 3 and 13. Our approach to dialogue management, described in Section 6, supports an elegant and robust handling of correction moves.

1.1 A note on versions

The lengthy certification and sign-off process required by NASA means that software actually deployed in space typically lags behind the latest development version, and for these reasons the version of Clarissa described in this paper differs in some respects from the one currently installed on the ISS. In general, our focus is on presenting what we think are the best solutions to the design problems we have encountered, rather than on strict historical accuracy.

2 System Overview

The Clarissa system consists of a set of software modules, written in several different languages, which communicate with each other through the SRI Open Agent Architecture [Martin et al., 1998]. Commands can be issued either using voice, or through the GUI. This section gives an overview of the system as a whole. We start by listing the different types of supported functionalities

(Section 2.1), and then describe the main modules (Section 2.2) and the information flow between them (Section 2.3).

2.1 Supported functionality

The system supports about 75 individual commands, which can be accessed using a vocabulary of about 260 words. Many commands can also be carried out through the GUI. The main system functionality is as follows. In each case, we briefly describe the type of functionality, and give examples of typical commands.

- Navigation: moving to the following step or substep ("next", "next step", "next substep"), going back to the preceding step or substep ("previous", "previous substep"), moving to a named step or substep ("go to step three", "go to step ten point two").
- Visiting non-current steps, either to preview future steps or recall past ones ("read step four", "read note before step nine"). When this functionality is invoked, the non-current step is displayed in a separate window, which is closed on returning to the current step.
- Opening and closing procedures ("open E M U checkout procedure", "close procedure"). It is possible to have multiple procedures open simultaneously, and to move between them freely.
- Recording, playing and deleting voice notes ("record voice note", "play voice note on step three point one", "delete voice note on substep two").
- Setting and cancelling alarms ("set alarm for five minutes from now", "cancel alarm at ten twenty one").
- Showing or hiding pictures ("show figure two", "hide the picture").
- Changing the TTS volume ("increase volume", "quieter").
- Querying status ("where are we", "list voice notes", "list alarms").
- Commands associated with "challenge verify mode". This is a special mode suitable for particularly critical parts of a procedure, which aggressively asks for confirmation on each step. The user can directly enter or exit this mode ("enter challenge verify mode", "exit challenge verify mode"), or else set challenge verify mode on a specified step or range of steps ("set challenge verify mode on steps three through twelve"). When in challenge verify mode, the user must explicitly confirm that each step has been completed ("verified", "step three complete"); if they fail to do so, the system enters into a confirmation subdialogue. Other subdialogues are entered if the user attempts to skip steps or move backwards in the procedure. The system keeps a database of completed, uncompleted and skipped steps, which can be accessed by commands like "list completed" or "list skipped".
- Responding to system questions. Most of the dialogue is user-initiative, but the system can enter short information-seeking subdialogues in certain situations. There are three main types of system question:
  - Yes/no questions. The user can respond with a yes/no word ("yes", "affirmative", "no", "negative").

  - First choice/second choice questions. These are used at some branch points in procedures. The user can respond "first choice" or "second choice".
  - Numerical value queries. The user can respond with a specific value ("zero", "eleven", "eight thousand two hundred four", "sixty one point five"), or give a null response ("no value").

- Undoing and correcting commands. Any command can be undone ("undo", "go back") or corrected ("no I said increase volume", "I meant exit review mode"). In some cases, the command can be expressed elliptically ("no I said three point one", "I meant step four").

2.2 Modules

The main software modules of the system are the following:

Speech Processor Handles low-level speech recognition and speech output.

Semantic Analyser Converts output from the speech processor into abstract dialogue move form.

Response Filter Decides whether to accept or reject the user's spoken input.

Dialogue Manager Converts abstract dialogue moves into abstract dialogue actions, and maintains discourse context.

Output Manager Converts abstract dialogue actions into concrete responses.

GUI Mediates conventional screen and keyboard based interaction with the user.

We describe each of them in more detail in the rest of this section. Interactions between the modules are described in Section 2.3.

2.2.1 Speech Processor

The Speech Processor module is built on top of the commercial Nuance Toolkit platform [Nuance, 2003], and is implemented in C++ using the RecEngine API. The top-level functionalities it provides are speech recognition and spoken output. Speech recognition can be carried out using either a grammar-based or a statistical language model; speech output can be either through playing recorded wavfiles, or by using a TTS engine.

The Speech Processors's output from speech recognition always includes a list of words, each tagged with a confidence score. When using a grammar-based language model, it also includes a logical form representation defined by the grammar. Note that the grammar's "logical forms" and the dialogue manager's "dialogue moves" are *not* the same. Grammar-based language models are derived from a general linguistically motivated unification grammar, using methods described in Sections 4.1 and 4.3. Statistical language models are trained using standard methods.

The main technical challenge when building the Speech Processor module was to make all processing fully asynchronous, and provide fine-grained control over "barge-in" (interruptions by the user while the system is talking). The default method provided by Nuance for dealing with this situation is "start of speech barge-in"; when the user interrupts the system, it immediately stops talking. This is however completely inappropriate for a system which uses an open microphone

processing model, since the system should only stop talking if it decides that input is meaningful and directed towards it. The Speech Processor module is thus configured so as to stop talking only in response to an explicit abort signal, sent from the Dialogue Manager. The details are presented a little later, in Section 2.3.

2.2.2 Semantic Analyser

The Semantic Analyser is implemented in SICStus Prolog. It receives the output of the Speech Processor (a string of words, possibly combined with a logical form), and converts it into a dialogue move. The methods used to do this combine hand-coded patterns and corpus-derived statistical information, and are described in Sections 4.2 and 4.3. Figure 6 shows examples of the different levels of representation, contrasting logical forms with dialogue moves.

2.2.3 Response Filter

Since recognition is carried out in open microphone mode, at least some of the speech input needs to be rejected. This function is performed by the Response Filter. The Response Filter receives the surface input from the recogniser (a list of words tagged with confidence scores), and produces a binary judgement, either to accept or to reject. It is implemented in C using Support Vector Machine techniques described in Chapter 5.

2.2.4 Dialogue Manager

The Dialogue Manager is implemented in SICStus Prolog. It accepts dialogue moves from the Semantic Analyser and the Output Manager, and produces a lists of abstract dialogue actions as output. It also maintains a dialogue state object, which encodes both the discourse state and the task state. The Dialogue Manager is entirely side-effect free, and its operation is specified by a declarative *update function*

$$f : State \times Move \rightarrow State \times Actions$$

This is described further in Section 6.

2.2.5 Output Manager

The Output Manager accepts abstract dialogue actions from the Dialogue Manager, and converts them into lists of procedure calls. Executing these calls results in concrete system responses, which can include production of spoken output, sending display requests to the GUI, and sending dialogue moves back to the Dialogue Manager. The Dialogue Manager is implemented in SICStus Prolog.

2.2.6 GUI

The GUI is written in Java Swing, and mediates normal keyboard and screen based interaction with the user. It accepts input from the Output Manager in the form of display requests. It can also convert keyboard input from the user into dialogue moves, which are sent to the Dialogue Manager.

2.3 Information flow

Section 2.2 describes the software modules at the level of functionality. For reasons of efficiency, they are not all implemented as separate OAA agents, since OAA message passing introduces a tangible overhead. Instead, the modules are combined into three larger OAA agents, as follows:

Speech and response filter agent This agent combines the two software modules written in C-derived languages, namely the Speech Processor and Response Filter.

Semantic analysis, dialogue and output manager This agent combines the three Prolog-based agents: the Semantic Analyser, the Dialogue Manager, and the Output Manager.

GUI agent The GUI module forms a standalone agent.

The following example illustrates the flow of control when responding to a simple voice command. The system has got as far as step 2.1 in the current procecure, and the user now speaks the command "go to step three point two". The Speech Processor produces two outputs, a surface result consisting of a list of words annotated with confidence scores:

```
go:71 to:65 step:62 three:65 point:56 two:16
```

and a "logical form":

```
[[imp,
  form(imperative,
       [[go,
         term(pro,you,[]),
         [to,term(null,
                  step,
                  [[number,[decimal,3,2]]])]]])]]
```

The surface result is passed to the Response Filter, resulting in an "accept" classification. The Response Filter makes this decision using a Support Vector Machine trained on several thousand labelled examples: in this case, its "justification" for accepting is mostly the high average confidence score, and to a less extent the common words and bigrams. The list of words and the logical form are now passed to the Semantic Analyser, which converts them into the "dialogue move" representation

```
go_to([decimal,3,2])
```

Note the difference between the logical form and the dialogue move: the logical form is a linguistically motivated representation derived from the general base grammar, while the dialogue move is a domain-specific representation.

The dialogue move is now passed to the Dialogue Manager, which invokes an "update rule" acting on the dialogue move and the previous dialogue state object S_{in}, a list of key/value pairs. The result is a new dialogue state S_{out}, and an abstract action which consists of reading out the contents of step 3.2. The Dialogue Manager then computes which other actions it needs to perform by comparing the states S_{in} and S_{out}. In this case, the only relevant difference is in the `location` component, which marks the current step in the procedure; the values in S_{in} and S_{out} are `[2, 1]` and `[3, 2]` respectively. Since the values are different, the Dialogue Manager adds a second

abstract action, which causes the GUI to scroll from step 2.1 to 3.2. The processing involved here is described in more detail in Section 6.

The two abstract actions (a reading action, and a scrolling action), are passed to the Output Manager, which converts them into concrete procedure calls. When these calls are executed, the first one results in sending a message to the Speech Processor agent to play a list of recorded wavfiles; the second results in a message to the GUI, which causes it to scroll from 2.1 to 3.2.

It would also have been possible to issue the same command through the GUI, by opening the "Move" dialogue box and entering the number of the step on the keyboard. In this case, the GUI starts by sending the same dialogue move,

```
go_to([decimal,3,2])
```

to the Dialogue Manager. Subsequent processing is exactly the same as if the command had been issued through the voice modality. In particular, irrespective of the modality of the original command, the user would be able to correct it by voice, for example by saying "undo" or "no i meant go to three point three".

3 Writing voice-navigable documents

The problem of representing written documents for spoken use can be surprisingly complex. In the case of Clarissa, several important and potentially conflicting constraints had to be satisfied by the design of the procedure representation. First, since ISS procedures are critical formal documents that typically reflect hundreds or even thousands of person-hours of effort, including a lengthy approval process, it is not feasible to replace them with a new structure. Second, although the associated visual display is required to faithfully reproduce the official procedures, reading out the procedures verbatim is unworkable. Third, the NASA procedure writing community must accept the spoken version as equivalent in content to the original procedure, or it cannot be formally approved as a valid substitute in the safety-critical environment of the ISS. And finally, all procedure specific information must be incorporated into the procedure representation, rather than the browser code, to enable use of the same procedure browser for many procedures.

Our approach to satisfying these contraints represents procedures in an XML format that contains all the text and layout information present in the original written procedure, together with additional information which specifies how the text is to be read out in the context of procedure execution. For each procedure, an XML file is compiled into an HTML display document which will exactly mimic the appearance of the original paper document, and also an annotated structure that can be followed by the dialogue manager and which will permit the text to be augmented and paraphrased where appropriate to enable it to be read aloud in a natural manner.

The browser treats this compiled XML procedures as data. This makes it possible to drop in an updated procedure without re-compiling the entire Clarissa system. Clarissa currently handles five International Space Station procedures. These procedures are fairly elaborate; they average approximately 53 steps each and require an average of 980 lines of XML to represent them.

Given that the spoken version can, and in a number of cases must, differ from the original written version of the procedure, we are compelled to address the question of how much, and in what ways the versions can differ. A basic design principle in Clarissa is that the spoken version models what a human would read aloud while using the document to do a task. In some parts of the procedures the written and spoken versions are the same and in others they diverge. The

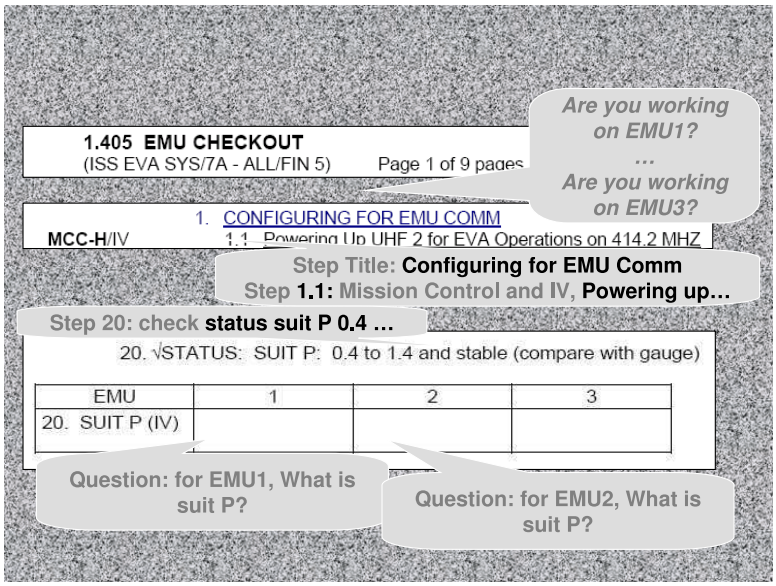

Figure 4: Adding voice annotations to a table

divergences are the major source of complexity in the XML representation. These divergences arise from basic differences between the modalities and perhaps even more crucially from the fact that the spoken version must be adequate for possible values of the document's dynamic content as well as the document's use.

In some cases, the differences are minor: wording for fluent speech often differs from highly abbreviated and/or acronym filled text. For example: "H20 vlv ↔ MSB" would read better as "water valve, disconnect from micro-sample bag". In other cases, visual and spoken structures are so different that even if one wanted to read that part of the document verbatim, it would not be clear how to do it. Tables are a clear example. Visually, a table provides information in the formatting. One can scan top and side headers to understand what the values in a table cell mean or what kind of material should be filled in to the table cell. Lines typically individuate the cells. What should a spoken version of a table be like? How do you "read" lines separating cells, or "read" the spatial layout of the table? A human reading aloud would probably not give details of the formatting or read all the headers but would present the table information in a way motivated by how the table needed to be used. Was the table to be read out, or filled in? In what order should cells be read?

Figures 4 and 5 illustrate the relationship between the written and spoken versions using the procedure fragments that we have already seen in the dialogue from Figures 1, 2 and 3. Black text in the speech bubbles indicates material that is unchanged from the written version. Grey text represents material added or paraphrased by the dialogue system. Figure 4 illustrates a case in which a table is used to elicit values from the user and record them. In this procedure maintenance is done on between one and three space suits (EMUs). At the beginning of the procedure the spoken system adds queries about which EMUs will be used so that it can restrict its later activity to only the relevant columns of the table. To fill in the table, a human reader would likely elicit the values by incorporating the column and row header information into a query or directive for

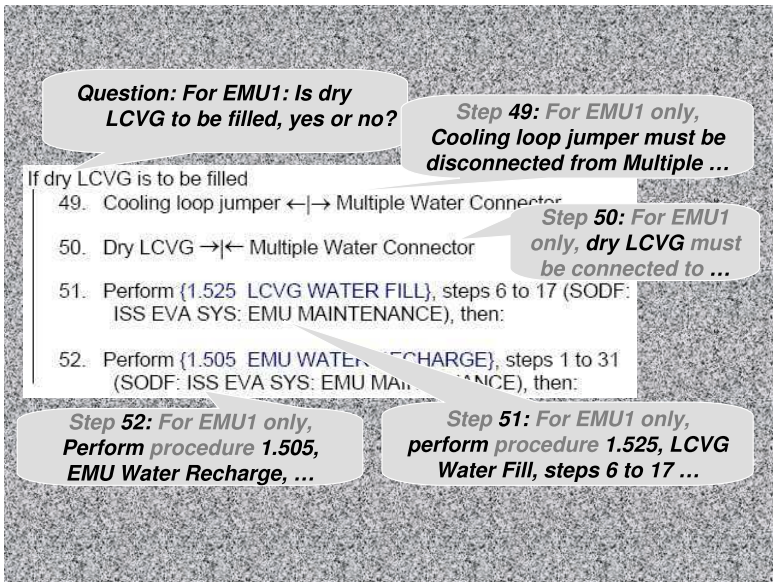

Figure 5: Adding voice annotations to a group of steps

each cell. The system follows this strategy, associating a query to the user with each cell. Once the user answers, the information is displayed in the relevant cell of the table.

Figure 5 illustrates another variant on initial collection of information for use in customising behavior in later steps. In this case the system queries the user on which EMUs will have a component, the LCVG, filled during the procedure. This information from the user allows the system to construct the appropriate step prefix *For EMU1 only*, and traverse only the EMU1 column of tables in the LCVG Water Fill Procedure, if the user chooses at step 51 to jump to that procedure. Note in both this example and in Figure 4 that language must be added for reading out symbols, and some of the abbreviations and acronymn. Also, words such as *procedure*, in Figure 5 are added for spoken naturalness and clarity.

3.1 Structure of the procedure XML representation

The types of divergences between the written and spoken versions discussed above have motivated development of appropriate XML structures to encode them. The most interesting cases are the following:

"Invisible" and "inaudible" constructs: The spoken and visual versions of a procedure can differ for a particular item. In other cases, there could be material that should only be spoken or only be displayed. Attributes are used in the XML to mark material that should be displayed and not spoken ("inaudible") or spoken and not displayed ("invisible"). For example, all the queries associated with table cells are invisible as are the questions to the user at about how which EMUs are being processed.

Conditional text and speech: Some steps apply only when certain conditions are met. This structure supports selecting a subset of procedure steps to execute based on various conditions.

If a visible step is conditional, and its conditions are not met, the visual display grays it out, and it is not read.

Eliciting values: Written procedures have implicit and explicit conditions based on values that a human user will have access to, but a computer system may not. For example, questions about which items will be processed, or queries to the user for values to be fill in to a table. The ValueStep XML is structure designed to elicit values from the user. When a value is obtained from the user in a ValueStep, the user's response is assigned to a variable. The minimum and maximum approved values are tested by the system and the system is able to warn the user if a value is out of range.

Tables: The use of a table in executing a procedure is to collect the relevant values and record them in the appropriate cells. For each table cell, there is an associated question to the user eliciting the appropriate value. Each table cell question is part of an invisible ValueStep. The structures AddendumList and Addendum implement the inclusion of additional spoken information about a value beyond the min and max. For example, low value out of range may necessitate the user contact mission control while a high value might not require this.

SpeechBefore: Supports material to be spoken in the introduction of a step, before the step number. In Figure 5 SpeechBefore is used as a verbal marker of the scope of the initial "If dry LCVG is to be filled". It is set in the example to *For EMU1 only*

Setting variables: Used to set variables needed in execution of steps. Variables are used to store and manipulate values collected from the user. Variables are also used to control conditional text and speech behavior.

3.2 Representing task-related discourse context

As we have already seen, procedures will frequently need to encode requests to acquire information from the user. This information will then become part of the task-related discourse context, and is stored as the bindings of variables defined in the procedure. Later steps may access the variable bindings so as to make decisions about conditional steps, using suitable XML constructs.

The use of context variables creates problems when combined with the requirement that users should be able to move around freely in the procedure. Suppose that a value for variable V is acquired at step 5, and is then used as part of a condition occurring in step 10. If the user skips straight from step 4 to step 10, V will be unbound, and the condition will not be evaluable. In situations like these, we would like the dialogue manager to be able to alert the user that they need to execute the earlier step (in this case, step 5) before moving to step 10.

In the general case, this problem appears to be quite intractable; use of a sufficiently expressive variable-setting language can require arbitrarily complex reasoning to determine which steps need to be executed in order to give a binding to a specific variable. A workable solution must involve restricting the way in which variables may be set. The solution we have implemented is fairly drastic, but was sufficient for the purposes of the initial Clarissa project: for each variable V, we only permit V to be assigned a value in at most one procedure step, so that it is possible to associate V with the step in which it can potentially receive a value. If a jump results in attempted evaluation of an unbound variable, the dialogue manager can then tell the user that they first need to execute the relevant earlier step. An example of this behaviour can be seen in turn $\mathbf{S}_4$ in Figure 1.

3.3 The design and approval process

The critical requirement when creating a voice version of a procedure is that the result should in a strong sense be the same document as the original. The procedure writing community at NASA is very focussed on safety considerations, and any new version of a procedure has to go through a lengthy sign-off procedure. Extending this procedure to include voice documents has been one of the unexpected challenges of the Clarissa project.

During the design phase, our experience has been that it is unproductive to show the XML directly to the procedure writers, who generally do not have a background that makes it easy for them to work with this kind of format. Instead, we split the document into independent modules, and for each module write down scenarios exhaustively describing all possible ways in which the document module can be read out by the browser. Since the browser is interactive, these scenarios are in the form of interactive dialogues between the user and the system. The scenarios serve as specification documents in the interaction between procedure writers and voice system implementers, and are refined until they converge to a mutually agreed solution.

Using such written scenarios currently imposes practical restrictions on the types of structures that can be used in procedure documents, since overly general constructions are not amenable to exhaustive description in this form. Additionally, these scenarios are difficult to keep up to date when requirements and implementation details force changes. An interesting question for future research is how to develop ways to generate such scenario documents that both permit more expressive communications between document writers and speech developers and that are amenable to automation. Having a tool to automatically generate expected system dialogue for common scenarios would also benefit Quality Assurance teams who need to validate such spoken dialogue system code. For the Clarissa system, such validation is now done using manual testing, but this will not scale acceptably to a system that needs to handle thousands of possible procedure documents.

4 Grammar-based recognition

As described in Section 2, Clarissa uses a grammar-based recognition architecture. At the start of the project, we had several reasons for choosing this approach over the more popular statistical one. First, we had no available training data. Second, the system was to be designed for experts who would have time to learn its coverage; although there is not a great deal to be found in the literature, an earlier study in which we had been involved [Knight et al., 2001] suggested that grammar-based systems outperformed statistical ones for this kind of user. A third consideration that affected our choice was the importance of "open mic" recognition; there was again little published research, but folklore results suggested that grammar-based recognition methods had an edge over statistical ones here too.

Obviously, none of the above arguments are very strong. We thus wanted to implement a framework which would allow us to compare grammar-based methods with statistical ones in a methodologically sound way, and also retain the option of switching from a grammar-based framework to a statistical one if that later appeared justified. The Regulus [Rayner et al., 2003, Regulus, 2005] and Alterf [Rayner and Hockey, 2003] platforms, which we have developed under Clarissa and other earlier projects, are designed to meet these requirements.

The basic idea behind the Regulus platform is to construct grammar-based language models

using example-based methods driven by small corpora. Since grammar construction is now a corpus-driven process, the same corpora can also be used to build normal statistical language models, facilitating a direct comparison between the two methodologies. On its own, however, Regulus only permits comparison at the level of recognition strings. Alterf extends the paradigm to the semantic level, by providing a uniform trainable semantic interpretation framework which can work on either surface strings or logical forms. Sections 4.1 and 4.2 provide an overview of Regulus and Alterf respectively, and Section 4.3 describes how we have used them in Clarissa.

Exploiting the special properties of Regulus and Alterf, Section 4.4 presents an evaluation of Clarissa's speech understanding component, including a methodologically sound comparison between the implemented grammar-based architecture and a plausible statistical/robust counterpart. In terms of speech understanding on in-domain utterances, our bottom-line conclusion is that, at least for this application, grammar-based methods significantly outperform statistical ones. Section 5 goes on to show that grammar-based methods also offer significant advantages for the task of detecting cross-talk.

4.1 Regulus

The core functionality provided by the Regulus Open Source platform is compilation of typed unification grammars into annotated context-free grammar language models expressed in Nuance Grammar Specification Language (GSL) notation [Nuance, 2003]. GSL language models can be converted into runnable speech recognisers by invoking the Nuance Toolkit compiler utility, so the net result is the ability to compile a unification grammar into a speech recogniser.

Experience with grammar-based spoken dialogue systems shows that there is usually a substantial overlap between the structures of grammars for different domains. This is hardly surprising, since they all ultimately have to model general facts about the linguistic structure of English and other natural languages. It is consequently natural to consider strategies which attempt to exploit the overlap between domains by building a single, general grammar valid for a wide variety of applications. A grammar of this kind will probably offer more coverage (and hence lower accuracy) than is desirable for any given specific application. It is however feasible to address the problem using corpus-based techniques which extract a specialised version of the original general grammar.

Regulus implements a version of the grammar specialisation scheme which extends the Explanation Based Learning method described in [Rayner et al., 2002]. There is a general unification grammar, loosely based on the Core Language Engine grammar for English [Pulman, 1992], which has been developed over the course of about ten individual projects. The semantic representations produced by the grammar are in a simplified version of the Core Language Engine's Quasi Logical Form notation [van Eijck and Moore, 1992].

A grammar built on top of the general grammar is transformed into a specialised Nuance grammar in the following processing stages:

1. The training corpus is converted into a "treebank" of parsed representations. This is done using a left-corner parser representation of the grammar.

2. The treebank is used to produce a specialised grammar in Regulus format, using the EBL algorithm [van Harmelen and Bundy, 1988, Rayner, 1988].

3. The final specialised grammar is compiled into a Nuance GSL grammar.

4.2 Alterf

Alterf is another Open Source toolkit, whose purpose is to allow a clean combination of rule-based and corpus-driven processing in the semantic interpretation phase. Alterf characterises semantic analysis as a task slightly extending the "decision-list" classification algorithm [Yarowsky, 1994, Carter, 2000]. We start with a set of *semantic atoms*, each representing a primitive domain concept, and define a semantic representation to be a non-empty set of semantic atoms. For example, in Clarissa we represent the utterances

> please speak up
> set an alarm for five minutes from now
> no i said go to the next step

respectively as

> {`increase_volume`}
> {`set_alarm, 5, minutes`}
> {`correction, next_step`}

where `increase_volume`, `set_alarm`, `5`, `minutes`, `correction` and `next_step` are semantic atoms. As well as specifying the permitted semantic atoms themselves, we also define a *target model* which for each atom specifies the other atoms with which it may legitimately combine. Thus here, for example, `correction` may legitimately combine with any atom, but `minutes` may only combine with `correction`, `set_alarm` or a number.

Training data consists of a set of utterances, in either text or speech form, each tagged with its intended semantic representation. We define a set of *feature extraction rules*, each of which associates an utterance with zero or more features. Feature extraction rules can carry out any type of processing. In particular, they may involve performing speech recognition on speech data, parsing on text data, application of hand-coded rules to the results of parsing, or some combination of these. Statistics are then compiled to estimate the probability $p(a \mid f)$ of each semantic atom a given each separate feature f, using the standard formula

$$p(a \mid f) = (N_f^a + 1)/(N_f + 2)$$

where N_f is the number of occurrences in the training data of utterances with feature f, and N_f^a is the number of occurrences of utterances with both feature f and semantic atom a.

The decoding process follows [Yarowsky, 1994] in assuming complete dependence between the features. Note that this is in sharp contrast with the Naive Bayes classifier [Duda et al., 2000], which assumes complete *independence*. Of course, neither assumption can be true in practice; however, as argued in [Carter, 2000], there are good reasons for preferring the dependence alternative as the better option in a situation where there are many features extracted in ways that are likely to overlap.

We are given an utterance u, to which we wish to assign a representation $R(u)$ consisting of a set of semantic atoms, together with a target model comprising a set of rules defining which sets of semantic atoms are consistent. The decoding process proceeds as follows:

1. Initialise $R(u)$ to the empty set.

2. Use the feature extraction rules and the statistics compiled during training to find the set of all triples $\langle f, a, p \rangle$ where f is a feature associated with u, a is a semantic atom, and p is the probability $p(a \mid f)$ estimated by the training process.

3. Order the set of triples by the value of p, with the largest probabilities first. Call the ordered set T.

4. Remove the highest-ranked triple $\langle f, a, p \rangle$ from T. Add a to $R(u)$ iff the following conditions are fulfilled:

   - $p \geq p_t$ for some pre-specified threshold value p_t.
   - Addition of a to $R(u)$ results in a set which is consistent with the target model.

5. Repeat step (4) until T is empty.

Intuitively, the process is very simple. We just walk down the list of possible semantic atoms, starting with the most probable ones, and add them to the semantic representation we are building up when this does not conflict with the consistency rules in the target model. We stop when the atoms suggested are too improbable, that is, have probabilities below a specified threshold.

4.3 Using Regulus and Alterf in Clarissa

We now describe the details of how we have used the Regulus and Alterf platforms in Clarissa. The Clarissa Regulus grammar is composed of the general Regulus grammar and the general function-word lexicon, together with a Clarissa-specific domain lexicon containing 313 lemmas, which realise 857 surface lexical rules. (Most of the difference is accounted for by verbs, which in the Regulus grammar have five surface forms). Table 1 summarizes the data for the domain-specific lexicon.

POS	#Entries		Examples	Example contexts
	Lemmas	Words		
Verb	129	645	continue go put	"**continue**" "**go** to step three" "**put** a voice note on step six"
Noun	99	127	caution alarm mode	"read **caution** before step eleven" "list **alarms**" "enter challenge verify **mode**"
Number	25	25	zero fiver	"set alarm for ten **zero fiver**"
Interjection	20	20	copy	"**copy** go to step three"
Preposition	15	15	on	"give me help **on** navigation"
Adjective	15	15	skipped	"list **skipped** steps"
Adverb	10	10	louder	"speak **louder**"
Total	313	857		

Table 1: Summary information for Clarissa lexicon. For each part of speech, we give the number of lexicon entries, example words, and example contexts.

As sketched in Section 4.1, the grammar is converted into a specialised form using an example-based method driven by a training corpus, and then compiled into a CFG language model. The training corpus we use for the current version of the system contains 3297 examples; of these, 1349 have been hand-constructed to represent specific words and constructions required for the application, while the remaining 1953 are transcribed examples of utterances recorded during system development.

The parameters guiding the grammar specialisation process have been chosen to produce a fairly "flat" grammar, in which many noun-phrases become lexical items. This reflects the generally stereotypical nature of language in the Clarissa domain. The specialised unification grammar contains 491 lexical and 162 non-lexical rules; Table 2 shows examples of specialised grammar rules, together with associated frequencies of occurrence in the training corpus. The specialised grammar is compiled into a CFG language model containing 427 non-terminal symbols and 1999 context-free productions. Finally, the training corpus is used a second time to perform probabilistic training of the CFG language model using the Nuance `compute-grammar-probs` utility, and the resulting probabilistic version of the language model is compiled into a recognition package using the `nuance-compile` utility.

On a 1.7GHz P4 laptop running SICStus 3.8.5, the whole compilation process takes approximately 55 minutes. This divides up as about 30 minutes to parse the training corpus, 10 minutes to extract the specialised grammar from the parsed treebank, 10 minutes to perform unification grammar to CFG language model compilation, and 5 minutes for probabilistic grammar training and Nuance recogniser compilation.

Rule	Freq	Example
`S --> V NP`	606	"[delete] [the voice note]"
`NP --> NUMBER`	511	"[one hundred thirty seven]"
`ADJ --> next`	508	"no i said [next]"
`NP --> step NUMBER`	481	"play voice note on [step four]"
`SIGMA --> INTERJECTION NP`	344	"[no i meant] [four point one]"
`S --> V P NP`	321	"[go] [to] [row one]"
`S --> V NP POST_MODS`	295	"[set] [timer] [for two minutes]"
`NUMBER --> NUMBER point NUMBER`	278	"[twenty five] [point] [nine]"
`S --> V`	259	"[speak up]"
`POST_MODS --> P NP`	228	"set alarm [at] [three zero six]"
`V --> go back`	108	"[go back]"
`S --> V NP NP`	78	"[show] [me] [figure one]"
`NP --> the voice note`	40	"cancel [the voice note]"
`S --> V P NP POST_MODS`	28	"[go] [to] [the note] [before step one]"
`NP --> what procedure`	12	"[what procedure] are we on"
`S --> NP V NP`	11	"[what] [is] [the current step]"
`V --> exit`	6	"[exit] review mode"
`NP --> value`	3	"[value] is undefined"

Table 2: Examples of rules in specialised version of Clarissa grammar. For each rule we give the context-free skeleton, the frequency of occurrence in the training corpus, and an example.

Surface
"no i said go to step five point three"

Logical form

```
[[interjection, correction],
 [imp,
  form(imperative,
       [[go,
         term(pro, you, []),
         [to, term(null, step, [[number, [decimal,5,3]]])]]])]]
```

Alterf output

```
[correction, go_to, [decimal,5,3]]
```

Dialogue move

```
correction(go_to([decimal,5,3]))
```

Figure 6: Examples showing different levels of representation for a Clarissa utterance. We show the surface words, the general logical form produced by the Regulus grammar and derived recogniser, the list of semantic atoms produced by Alterf, and the dialogue move.

Semantic representations produced by the Clarissa grammar are general domain-independent logical forms. By construction, the same representations are produced by the specialised grammar and the derived recogniser. The Alterf package is used to convert these general representations into unordered lists of semantic atoms; a final post-processing stage transforms Alterf output into the "dialogue moves" used as input by the dialogue manager. Figure 6 shows examples of these different levels of representation.

Recall that the Alterf algorithm requires definition of feature extraction rules, so that it can then trained to acquire associations between extracted features and cooccurring semantic atoms. We have experimented with three different kinds of feature extraction rules: surface N-grams, hand-coded surface patterns and hand-coded logical form patterns.

Surface N-gram features are the simplest type. First, the recognised string is tokenised using a phrase-spotting grammar that identifies numbers, times, durations in minutes and seconds, and a few other similar types of expression. Bigrams and trigrams are then extracted from the tokenised string. Thus for example the string "go to step five point three" would be tokenised as

```
[*start*, go, to, step, [decimal,5,3], *end*]
```

and yields the trigrams

```
[*start*, go, to]           [go, to, step],
[to, step, [decimal,5,3]]   [step, [decimal,5,3], *end*]
```

When calculating associations, Alterf replaces numbers and similar constructs with generic tokens: after this replacement, the trigrams above become

```
[*start*, go, to]              [go, to, step],
[to, step, *decimal_number*]   [step, *decimal_number*, *end*]
```

Thus in this case Alterf will not derive an association between the trigram `[to, step, [decimal,5,3]]` and the semantic atom `[decimal,5,3]`, but rather between the generic trigram `[to, step, *decimal_number*]` and the generic semantic atom `*decimal_number*`.

Unsurprisingly, we discovered that surface N-gram features were not particularly reliable. (Figures are presented in Section 4.4 supporting this contention). We then implemented two more sets of feature extraction rules, which defined different types of hand-coded patterns. The first set consists of conventional phrasal patterns over the tokenised recognition string, written in a simple string-matching language; the second set encodes structural patterns in the logical form. Examples of string-based and logical-form-based patterns are shown in Figure 7. The version of Clarissa described here has 216 string-based patterns and 305 logical-form-based patterns. The patterns have been developed and debugged using the 3297 utterance training corpus: on this corpus, each set of patterns has a classification error rate of about 0.5%.

4.4 Evaluating speech understanding performance

Having described the speech understanding architecture, we now present an evaluation. There are a large number of types of experiment which we could potentially have carried out. Given limited resources, we decided to focus on two main questions:

- How does the Regulus grammar-based framework compare against a more conventional framework using a class N-gram language model and a set of phrase-spotting rules?

- How do different types of features compare against each other? In particular, are logical-form-based patterns more effective than string-based or N-gram patterns, and is it useful to combine several types of pattern?

The next issue to resolve is the choice of appropriate performance metrics and test data. Given that we are essentially interested in speech understanding performance, the obvious metric is semantic error rate, by which we will specifically mean the proportion of utterances which fail to receive a correct semantic interpretation after Alterf processing. For completeness, we will also present figures for Word Error Rate (WER) and Sentence Error Rate (SER).

The choice of appropriate test data raises interesting questions. Initially, we had decided to follow standard practice, and collect data from naive subjects who had had no previous exposure to the system. We did in fact perform a data collection of this kind, but comparison of the results with the small amount of data we had from prospective astronaut users revealed a serious mismatch. Astronauts are exceptionally intelligent and capable individuals, who had a strong motivation to learn to use the system; the naive subjects were normal people, with no particular reason to want to acquire the relevant skills. The performance figures reflected this imbalance, with the astronauts scoring enormously better than nearly all of the naive subjects.

Our conclusion was that data collected from naive subjects was not useful, and that a much better fit came from the data recorded by system developers during the course of the project. Although the developers know the system a little better than the astronaut users, our intuitive observation was that the difference was not large, and that the astronauts would probably catch

String-based patterns

```
% ‘‘decrease’’ or ‘‘reduce’’ followed by ‘‘volume’’
% indicates the atom decrease_volume
surface_pattern([decrease/reduce,’...’,volume], decrease_volume).

% ‘‘back’’ not following ‘‘go’’ and at the end
% indicates the atom previous_line
surface_pattern([not_word(go),back,’*end*’], previous_line).

% ’’put’’ followed by ‘‘voice note’’
% indicates the atom record_voice_note
surface_pattern([put,’...’,voice,note], record_voice_note).
```

Logical-form-based patterns

```
% ‘‘decrease’’ or ‘‘reduce’’ with object an NP whose head is ‘‘volume’’
% indicates the atom decrease_volume
lf_pattern([decrease,_,term(_,volume,_)],decrease_volume).
lf_pattern([reduce,_,term(_,volume,_)],decrease_volume).

% ‘‘back’’ used as an interjection
% indicates the atom previous_line
lf_pattern([interjection,back],previous_line,back).

% ‘‘put’’ with object an NP whose head is ‘‘voice_note’’
% indicates the atom record_voice_note
lf_pattern([put,_,term(_,voice_note,_),_],record_voice_note).
```

Figure 7: Examples of string-based and logical-form-based patterns used in Clarissa.

up after only a relatively short period of use[2]. The developers' superior knowledge of the system was also counterbalanced to some extent by the fact that several of them were not native American speakers, whereas all the relevant astronauts were.

We are of course aware that this is not an ideal situation from the point of view of experimental design, but the virtual impossibility of getting significant quantities of astronaut time forced us to adopt a compromise: this appeared to be the best one. The figures below are thus based on a sample of 8158 in-domain utterances (23369 words) collected and transcribed during the course of the project. By "in-domain", we mean here that the utterances expressed commands meaningful in the context of the Clarissa task, and that the system should ideally have responded to them; we do not necessarily imply that the utterances were all syntactically well-formed or within the coverage of the grammar. Behaviour on out-of-domain data, where the best response is to ignore the utterance, is considered in Section 5. The data had not previously been used for development purposes, and can reasonably be considered as unseen.

In order to compare the Regulus-based recogniser with a conventional architecture, we used the Nuance SayAnything© tool and the same 3297 utterance training set to build a standard class N-gram model. We defined three classes, for numbers, times, and objects that could be displayed using the "show" command. Raw recognition performance figures for the two recognisers, measured in terms of WER and SER, are shown in Table 3.

Recogniser	WER	SER
GLM	6.27%	9.79%
SLM	7.42%	12.41%

Table 3: WER and SER for the Regulus-based recogniser (GLM) and a conventional class N-gram recogniser (SLM) trained on the same data.

The main experimental results are presented in Table 4. Here, we contrast speech understanding performance for the Regulus-based recogniser and the class N-gram recogniser, using several different sets of Alterf features. For completeness, we also present results for simulated perfect recognition, i.e. using the reference transcriptions. We used six different sets of Alterf features:

N-grams: N-gram features only.

LF: Logical-form-based patterns only.

String: String-based patterns only.

String + LF: Both string-based and logical-form based patterns.

String + N-grams: Both string-based and N-gram features.

String + LF + N-grams: All types of features.

As can be seen, the GLM performs very considerably better than the SLM. The best SLM version, S-3, has a semantic error rate of 9.6%, while the best GLM version, G-4 has an error rate of 6.0%,

[2]None of the astronauts used the system for more than a total of one hour. We had originally expressed concern that this would not give them adequate time to learn to use it effectively; after watching a sample training session, we were forced to admit that we had underestimated their capabilities.

a relative improvement of 37%. Part of this is clearly due to the fact that the GLM anyway has better WER and SER than the SLM. However, Table 3 shows that the relative improvement in WER is only 15% (7.42% versus 6.27%), and that in SER is 21% (12.41% versus 9.79%). The larger improvement by the GLM version at the level of semantic understanding is most likely accounted for by the fact that it is able to use logical-form-based features, which are not accessible to the SLM version. Although logical-form-based features do not appear to be intrinsically more accurate than string-based features (contrast rows T-2 and T-3), the fact that they allow tighter integration between semantic understanding and language modelling is intuitively advantageous.

It is interesting to note that the combination of logical-form-based features and string-based features outperforms logical-form-based features alone (rows G-4 and G-5). Although the difference is small (6.0% versus 6.3%), a pairwise comparison shows that it is significant at the 1% level according to the McNemar sign test. There is no clear evidence that N-gram features are very useful. This supports the standard folk-lore result that semantic understanding components for command and control applications are more appropriately implemented using hand-coded phrase-spotting patterns than general associational learning techniques.

Version	Rec	Features	Errors		
			Rejected	Incorrect	Total
T-1	Text	N-grams	7.3%	5.9%	13.2%
T-2	Text	LF	3.1%	0.5%	3.6%
T-3	Text	String	2.2%	0.8%	3.0%
T-4	Text	String + LF	0.8%	0.8%	1.6%
T-5	Text	String + LF + N-grams	0.4%	0.8%	1.2%
G-1	GLM	N-grams	7.4%	9.7%	17.1%
G-2	GLM	LF	1.4%	4.9%	6.3%
G-3	GLM	String	2.9%	4.8%	7.7%
G-4	**GLM**	**String + LF**	**1.0%**	**5.0%**	**6.0%**
G-5	GLM	String + LF + N-grams	0.7%	5.4%	6.1%
S-1	SLM	N-grams	9.6%	11.9%	21.5%
S-2	SLM	String	2.8%	7.4%	10.2%
S-3	SLM	String + N-grams	1.6%	8.0%	9.6%

Table 4: Speech understanding performance for 8158 test sentences recorded during development, on 13 different configurations of the system. The "Rec" column indicates either simulated perfect recognition ("Text"), recognition using the Regulus-derived grammar-based language model ("GLM") or recognition using a class N-gram language model ("SLM"). The "Features" column indicates the Alterf features used.

Finally, Table 5 presents a breakdown of speech understanding performance, by utterance length, for the best GLM-based and SLM-based versions of the system. There are two main points we want to make here. First, speech understanding performance remains respectable even for the longer utterances; second, the performance of the GLM-based version is consistently better than that of the SLM-based version for all utterance lengths.

Length	#Utts	Best GLM (G-4)			Best SLM (S-3)		
		WER	SER	SemER	WER	SER	SemER
1	3049	5.7%	3.5%	2.5%	6.3%	4.2%	3.5%
2	1589	12.0%	12.0%	8.7%	14.6%	18.4%	14.6%
3	950	7.2%	12.8%	7.2%	10.4%	15.2%	15.4%
4	1046	7.6%	14.8%	9.9%	7.7%	15.6%	14.7%
5	354	5.7%	14.4%	9.0%	6.1%	19.8%	10.8%
6	543	2.8%	11.1%	7.2%	4.1%	15.3%	9.8%
7	231	3.0%	16.0%	3.5%	4.6%	19.5%	6.5%
8	178	4.4%	14.6%	4.5%	3.6%	16.3%	5.7%
9	174	3.9%	20.1%	9.2%	4.0%	20.7%	10.3%

Table 5: Speech understanding performance, broken down by utterance length, for the best GLM-based and SLM-based versions of the system (cf. Table 4). Results are omitted for the small group of utterances of length 10 or more.

5 Rejecting user speech

Section 4 presents figures describing the performance of the speech understanding component of the system, assuming that the task can be described as that of taking an input utterance constituting a valid system command, and assigning a correct semantic interpretation to it. This characterisation, however, omits an important dimension. At least some of the time, we know that input speech will not consist of valid system commands. The most common reason for this will be *cross-talk*: the user may break off addressing the system to converse with another person, but his[3] remarks will still be picked up the microphone and subjected to speech recognition. There is also the possibility that the user will say something that is outside the system's sphere of competence, either though carelessness or because they are uncertain about its capabilities.

We will refer to the choice between accepting and rejecting output from the speech recogniser as the "accept/reject decision". Usually, the speech recogniser produces a confidence score as part of its output, and the accept/reject decision is made simply by rejecting utterances whose confidence score is under a specified threshold. A recent example is [Dowding and Hieronymus, 2003], which reported an accuracy of 9.1% on cross-talk identification using the confidence threshold method.

In this rest of this section, we will show that adapted versions of standard kernel-based methods from the document classification literature can substantially improve on the baseline confidence threshold approach. First, however, we define the problem more precisely.

5.1 The Accept/Reject Decision Task

We first need to introduce suitable metrics for evaluating performance on the accept/reject task. We can define the following three categories of utterance:

Type A: Utterances directed at the system, for which a good semantic interpretation was produced.

[3]All the prospective astronaut users trained so far happen to be male.

Type B: Utterances directed at the system, and to which the system could in principle respond, but for which the semantic interpretation produced was incorrect. Usually, this is due to faulty recognition.

Type C: Utterances not directed at the system, or directed at the system but to which the system has no adequate way to respond.

We would like to accept utterances in the first category, and reject utterances in the second and third. If we want to measure performance on the accept/reject task, the most straightforward approach is a simple classification error rate. Ultimately, however, what we are most interested in is measuring performance on the top-level speech understanding task, which includes the recognition task, the semantic interpretation task, and the accept/reject task described here.

Constructing a sensible metric for the top-level task involves taking account of the fact that some errors are intuitively more serious than others. In a Type A error, the user can most often correct by simply repeating himself. In a Type B or C error, the user will typically have to wait for the system response, realise that it is inappropriate, and then undo or correct it, a significantly longer operation. This analysis suggests that errors should not all be counted equally, but rather be weighted to define a loss function. We also need to remember that some utterances produce no semantic interpretation at all. Obviously, these utterances can only be rejected. We consequently split class B into B0 (no interpretation) and B1 (bad interpretation), and class C into C0 (no interpretation) and C1 (irrelevant or bad interpretation).

It is not easy to give clear justifications for particular choices of weights. A reasonable candidate, which we will use in the rest of this paper, is given in Table 6. This reflects the observations that it generally appears to take about twice as long to recover from a false accept as a false reject, and that a false accept of a cross-talk utterance affects not only the user, but also the person with whom they are having the side conversation. Note that class B utterances (incorrect recognitions) are always errors, but carry a heavier penalty when accepted than when rejected. We divide the total weighted score over a given corpus by the maximum possible loss value, giving a normalised loss function whose value is always between 0 and 1.

Class	Score	
	Reject	Accept
A	1	0
B0	1	1
B1	1	2
C0	0	0
C1	0	3

Table 6: Weights for the top-level loss function. "Accept" and "Reject" refer to the decision made by the classifier; utterances in the classes B0 and C0 are always rejected, irrespective of the classifier's decision.

The discrepancy between the local loss function associated with the classifier and the task-level loss function raises the issue of how to align classifier objectives with task-level ones. In this initial work, we simplify the problem by decoupling the speech understanding and accept/reject subtasks, using separate metrics. Since the weighting on the task metric penalises false accepts more heavily

Positive

go:76 to:70 step:55 eleven:34
repeat:80 the:67 caution:80
stop:73 talking:65
no:36
i:49 meant:73 stop:68 reading:63
decrease:51 volume:89

Negative

stop:4
five:11
where:9 are:14 we:37
list:36 challenge:49 verify:38 it:10 was:65
mark:17 terse:65 mode:23 for:65 note:20 for:62 ten:20 thousand:41
column:7 eight:46 is:55 that:83

Figure 8: Examples of recognition results. Each word is tagged by a numerical confidence value. Positive hypotheses should be accepted, negative ones rejected.

than false rejects, we introduce an asymmetric loss function on the classifier score, which weights false accepts twice as heavily as false rejects. We will refer to this as the u_2 metric, and use it as the filtering task metric to compare different parameter settings.

5.2 An SVM-based approach

There are several information sources which could potentially be used as input to the accept/reject classification problem. So far, we have limited ourselves to the surface result returned by the Nuance recogniser, which consists of a list of words, each tagged by a numerical confidence value. Figure 8 shows examples of typical recognition results, grouped as "positive" (should be accepted), and "negative" (should be rejected).

As already noted, the usual way to make the accept/reject decision is through a simple threshold on the average confidence score; the Nuance confidence scores are of course designed for exactly this purpose. Intuitively, however, it should be possible to improve the decision quality by also taking account of the information in the recognised words. Looking through our data, we could see examples like the last one listed under "Negative" in Figure 8, which had high enough average confidence scores to be accepted, but contained phrases that were clearly very implausible in the context of the application. In the other direction, we noticed that the confidence scores for several common short utterances (in particular "yes" and "no") appeared for some reason to be artificially depressed; these utterances could safely be accepted with a lower confidence threshold than usual. We wanted a method that could identify and exploit patterns like these.

The accept/reject decision is clearly a kind of document classification problem. It is well known [Joachims, 1998] that margin-based classifiers, especially Support Vector Machines (SVM), get good results for this kind of task. Our original intuition, when beginning this piece of work, was that

standard SVM-based document-classification tools could be applied successfully to the accept/reject decision task; in practice, due to its stability and ready availability for research purposes, we used Joachims's SVM-light platform [Joachims, 2005]. There were two key issues we needed to address in order to apply SVM-light to the new task. First, we had to construct a suitable kernel function, which would define similarity between two recognition results; this kernel would need to reflect the fact that the objects were not text documents, but speech documents which had been passed through a noisy channel. Second, we had to take account of the asymmetric nature of the cost function.

5.2.1 Choosing a kernel function

As recognition results consist of sequences of words tagged with confidence scores, the kernel function must be based on this information. The simple bag of word representation, as traditionally used to represent written documents, loses the important confidence score information, and is unlikely to produce good results. Preliminary experiments not reported here confirmed that this was indeed the case.

We consequently modified the bag of words representation, so as to weight each word by its associated confidence score. This means the the recognition result is initially mapped into a vector in an V-dimensional space, where V is the vocabulary size; the component on each dimension is the confidence score on the relevant word, or zero if the word did not occur in the recognition result. If a word occurs multiple times, the confidence scores for each occurrence are summed; it is possible that a "max" or "min" operator would have been more appropriate, but multiple word occurrences only obtained in about 1% of the utterances. We also added an extra component to the vector, which represented the average of all the word confidence scores.

By using different kernel functions on the vector space representation, we were able to take into account various kinds of lexical patterns. In particular, nonlinear polynomial kernels of degree N encode unordered cooccurrences of N words (unordered gappy N-grams). In practice, we found that values of N higher than 2 gave no additional gains. We also experimented with string subsequence kernels [Lodhi et al., 2002], but these failed to improve performance either, while being computationally much more expensive. In the experiments reported below, we thus restrict ourselves to linear and quadratic kernels, representing unigram and unordered nonlocal bigram information.

5.2.2 Making the cost function asymmetric

There are at least two ways to introduce asymmetric costs in standard SVM implementations. Recall that SVM is optimizing a mixed criterion, which combines classification errors on a training set and a measure of complexity related to the margin concept ([Shawe-Taylor and Cristianini, 2004], p. 220). The simplest method is to penalize the distance to the margin for misclassified examples more highly for false positives than for false negatives. This can be done directly using the j parameter in the SVM-light implementation.

The drawback to the first approach is, however, that the algorithm is not really optimising the utility function, but a more or less related quantity [Navia-Vázquez et al., 2004]; this prompted us to investigate the use of calibration as well. Calibration [Bennett, 2003] aims at transforming SVM scores into posterior probabilities in a way that is independent from the class priors (basically $P(s(x) \mid Class)$ where $s(x)$ is the score associated with observation x). The optimal Bayesian

decision can then be adapted, once the new class priors are known ($P(Class)$), as well as error costs. For a binary problem (accept/reject) with equal cost of errors for all negative examples, when the class distribution can be assumed to be the same on both training and test sets, it is sufficient to approximate $P(Class = A \mid s(x))$, as the optimal Bayes decision is then based on minimizing the expected loss function.

In our case, the u_2 function penalises false accepts twice as heavily as false rejects: the optimal decision rule is thus to accept the utterance if

$$(2P(Class = B \text{ or } C \mid s(x))) < P(Class = A \mid s(x))$$

or, equivalently, if $P(Class = A \mid s(x)) > 2/3$. We used Isotonic Regression [Zadrozny and Elkan, 2002] to realize the mapping from SVM-scores into approximate posterior probabilities.

5.3 Experiments

A corpus of 10409 recorded and labelled spoken utterances was used in order to investigate the impact of three factors on classification and task performance:

Classifier We used three types of classifier: a simple threshold on the average confidence score; an SVM with a linear kernel; and an SVM with a quadratic kernel. The SVM classifiers used a set of features consisting of the average confidence score together with the weighted bag of words over the total vocabulary.

Asymmetric Error We used two different techniques to deal with asymmetric error costs: the j intrinsic parameter of SVM-light, and the recalibration procedure using Isotonic Regression. Recall that recalibration aims at optimising the u_2 loss function at SVM classification level, and not the task-level loss function. Without recalibration, the decision threshold on SVM-scores is 0.

Recognition We contrasted GLM and SLM methods, specifically using the best GLM-based recogniser (G-4) and the best SLM-based recogniser (S-3) from Table 4.

For each choice of parameters, we performed 10 random splits (training/test sets) of the initial set of labelled utterances, learned the model on the training sets, and evaluated the loss functions on the corresponding test sets. The final scores were obtained by averaging the loss functions over all 10 runs. Table 7 presents results for the most interesting cases.

5.4 Analysis of results

The following conclusions can be drawn from the figures in Table 7. Each of these conclusions was confirmed by hypothesis testing, using the Wilcoxon rank test, at the 5% significance level.

5.4.1 Improvement on baseline performance

The SVM-based method is very considerably better than the baseline confidence threshold method. The average classification error fell from 9.4% for the best baseline configuration (GT-1) to 5.5% for the best SVM-based configuration (GQ-3), a relative improvement of 42%. In particular, the false accept rate for cross-talk and out-of-domain utterances improved from 8.9% (close to the 9.1% cited

ID	Rec	Classifier	j	Error rates					
				Accept/reject classification					Task
				Classes			All	u_2	
				A	B	C			
ST-1	SLM	Threshold	1.0	5.5%	59.1%	16.5%	11.8%	15.1%	10.1%
SL-1	SLM	Linear	1.0	2.8%	37.1%	9.0%	6.6%	8.6%	7.4%
SL-2	SLM	Linear	0.5	4.9%	30.1%	6.8%	7.0%	8.1%	7.2%
SQ-1	SLM	Quad	1.0	2.6%	23.6%	8.5%	5.5%	7.0%	6.9%
SQ-2	SLM	Quad	0.5	4.1%	18.7%	7.6%	6.0%	7.0%	7.0%
SQ-3	SLM	Quad/r	1.0	4.7%	18.7%	6.6%	6.1%	6.8%	6.9%
GT-1	GLM	Threshold	0.5	7.1%	48.7%	8.9%	9.4%	10.7%	7.0%
GL-1	GLM	Linear	1.0	2.8%	48.5%	8.7%	6.3%	8.3%	6.2%
GL-2	GLM	Linear	0.5	4.7%	43.4%	6.0%	6.7%	7.9%	6.0%
GQ-1	GLM	Quad	1.0	2.7%	37.9%	6.8%	5.3%	6.7%	5.7%
GQ-2	GLM	Quad	0.5	4.0%	26.8%	6.0%	5.5%	6.3%	5.6%
GQ-3	GLM	Quad/r	1.0	4.3%	28.1%	4.7%	5.5%	6.1%	5.4%

Table 7: Performance on accept/reject classification and the top-level task, on 12 different configurations of the system. "Threshold" = simple threshold on average confidence; "Linear" = SVM classifier with linear kernel; "Quad" = SVM classifier with quadratic kernel; "Quad/r" = recalibrated version of SVM classifier with quadratic kernel; "j" = value of SVM-light j parameter; "A" = in-domain and correct semantic interpretation; "B" = in-domain and incorrect or no semantic interpretation; "C" = out-of-domain; "All" = standard classifier error rate over all data; "u_2" = weighted average of classifier error using u_2 weights; "Task" = normalised task metric score.

in [Dowding and Hieronymus, 2003]) to 4.7%, a 47% relative improvement, while the error rates on the other individual classes also improved. On the task performance metric, the improvement was from 7.0% to 5.4%, or 25% relative.

5.4.2 Kernel types

Quadratic kernels performed better than linear (around 25% relative improvement in classification error); however, this advantage is less marked when considering the task metric (only 3 to 9% relative increase). Though small, the difference is statistically significant. This suggests that meaningful information for filtering lies, at least partially, in the co-occurrences of groups of words, rather than just in isolated words.

5.4.3 Asymmetric error costs

We next consider the effect of methods designed to take account of asymmetric error costs (cf. Section 5.2.2). Comparing GQ-1 (no treatment of asymmetric error costs) with GQ-2 (intrinsic SVM-optimisation using the j-parameter) and GQ-3 (calibration), we see that both methods produce a significant improvement in performance. On the u_2 loss function that both methods aim to minimise, we attain a 9% relative improvement when using calibration and 6% when using intrinsic

SVM optimisation; on the task metric, these gains are reduced to 5% (relative) for calibration, and only 2% for intrinsic SVM-optimisation, though both of these are still statistically significant. Error rates on individual classes show that, as intended, both methods move errors from false accepts (classes B and C) to the less dangerous false rejects (class A). In particular, the calibration method manages to reduce the false accept rate on cross-talk and out-of-domain utterances from 6.8% on GQ-1 to 4.7% on GQ-3 (31% relative), at the cost of an increase from 2.7% to 4.3% in the false reject rate for correctly recognised utterances.

5.4.4 Recognition methods

Using the confidence threshold method, there was a large difference in performance between the GLM-based GT-1 and the SLM-based ST-1. In particular, the false accept rate for cross-talk and out-of-domain utterances is nearly twice as high (16.5% versus 8.9%) for the SLM-based recogniser. This supports the folklore result that GLM-based recognisers give better performance on the accept/reject task.

When using the SVM-based methods, however, the best GLM-based configuration (GQ-3) performs about as well as the best SLM-based configuration (SQ-1) in terms of average classification error, with both systems scoring about 5.5%. GQ-3 does perform considerably better than SQ-1 in terms of task error (5.4% versus 6.9%, or 21% relative), but this is due to better performance on the speech recognition and semantic interpretation tasks. Our conclusion here is that GLM-based recognisers do not necessarily offer superior performance to SLM-based ones on the accept/reject task, if a more sophisticated method than a simple confidence threshold is used.

5.4.5 Relative error rates on individual classes

A deeper look at the error rates by individual classes shows that, for all methods, by far the largest error rates are observed for the B category. This is not surprising: experience and intuition tell us that the hardest decision problem is distinguishing between correctly recognised in-domain and incorrectly recognised in-domain utterances. What makes it difficult is that "incorrect" is often better characterised as "partially correct", so there is much less information on which to base the decision.

For example, a problem in any domain which involves English numbers is the well-know "-teen/-ty" ambiguity — "thirty" sounds a lot like "thirteen", and so on. If the user says "go to step thirty" but the system recognises "go to step thirteen", the correct action is to reject, but there is little information available to support the decision: basically, we need confidence to be low on the word "thirty", but our observation is that this is atypical. There are a number of similar generic cases where partial recognition happens easily. We can contrast this with distinguishing between A and C class utterances — cross-talk recognition results tend to look very different from correct in-domain ones, since cross-talk recognition is usually completely wrong rather than just partially wrong. The decision problem is thus intuitively simpler.

5.5 Summary and conclusions

We have described a simple and general technique that allows standard SVM-based document classification methods to be applied to the accept/reject problem for speech understanding systems implemented on top of standard recognition platforms. This is particularly important for systems

that use open mic speech recognition. We have presented results of experiments carried out on a state-of-the-art command and control system, showing that our method gives substantial performance improvements compared to the standard method of using a threshold on the recognition confidence score.

We are currently extending the work described here in two different directions. First, we are implementing better calibration models, which in particular will allow us to relax the assumption that the class distributions are the same in the training and test data; second, we are investigating what further gains can be obtained from use of word-lattice recognition hypotheses and rational kernels [Haffner et al., 2003, Cortes et al., 2004]. We expect to be able to report on both of these issues in the near future.

6 Side-effect free dialogue management

A spoken language system that carries out a command and control task typically divides into three main components: the input module, the output module, and the dialogue manager. The input module processes spoken input, and converts it into abstract utterance representations using speech recognition, semantic analysis, and other techniques. The output module processes abstract action requests, and transforms them into concrete external actions, such as speaking, updating a visual display, or moving a real or simulated robot. Mediating between the input and output modules, we have the dialogue manager (DM), which at the most basic level transforms abstract utterance representations into abstract action representations.

Most spoken language systems have some notion of context, which typically will include the preceding dialogue, the current state of the task, or both. For example, consider the reaction of a simulated robot to the command "Put it on the block". This might include both remembering a recently mentioned object to serve as a referent for "it" (dialogue context), and looking around the current scene to find an object to serve as a referent for "the block" (task context). The DM will thus both access the current context as an input, and update it as a result of processing utterances.

In most dialogue systems, contextual information is distributed through the DM as part of the current program state. This means that processing of an input utterance involves at least some indirect side-effects, since the program state will normally be changed. If the DM makes procedure calls to the output module, there will also be direct side-effects in the form of exterior actions. As every software engineer knows, side-effects are normally considered a Bad Thing. They make it harder to design and debug systems, since they render interactions between modules opaque. The problem tends to be particularly acute when performing regression testing and evaluation; if a module's inputs and outputs depend on side-effects, it is difficult or impossible to test that module in isolation. The upshot for spoken language systems is that it is often difficult to test the DM except in the context of the whole system.

In this section, we will describe an architecture which directly addresses the problems outlined above, and which has been implemented in Clarissa. There are two key ideas. First, we split the DM into two pieces: a large piece, comprising nearly the whole of the code, which is completely side-effect free, and a small piece which is responsible for actually performing the actions. Second, we adopt a consistent policy about representing contexts as objects. Both discourse and task-oriented contextual information, without exception, is treated as part of the context object.

6.1 Side effect free dialogue management

Clarissa implements a minimalist dialogue management framework, partly based on elements drawn from the TRIPS [Allen et al., 2000] and TrindiKit [Larsson and Traum, 2000] architectures. The central concepts are those of *dialogue move*, *information state* and *dialogue action*. At the beginning of each turn, the dialogue manager is in an information state. Inputs to the dialogue manager are by definition dialogue moves, and outputs are dialogue actions. The behaviour of the dialogue manager over a turn is completely specified by an *update function* f of the form

$$f : State \times Move \rightarrow State \times Actions$$

Thus if a dialogue move is applied in a given information state, the result is a new information state and a set of zero or more dialogue actions. As in [Allen et al., 2000], the dialogue manager is bracketed between an *input manager* and an *output manager*. The input manager receives speech and other input directed to the dialogue manager, and transforms it into dialogue move format. The output manager takes dialogue actions produced by the dialogue manager, and transforms them into concrete sequences of procedure calls meaningful to other components of the system.

In the Clarissa system, most of the possible types of dialogue moves represent spoken commands. For example, `increase(volume)` represents a spoken command like "increase volume" or "speak up". Similarly, `go_to(step(2,3))` represents a spoken command like "go to step two point three". The dialogue move `undo` represents an utterance like "undo last command" or "go back" Correction utterances are represented by dialogue moves of the form `correction(X)`; so for example `correction(record(voice_note(step(4))))` represents an utterance like "no, I said record a voice note on step four". There are also dialogue moves that represent non-speech events. For example, a mouse-click on the GUI's "next" button is represented as the dialogue move `gui_request(next)`. Similarly, if an alarm goes off at time T, the message sent from the alarm agent is represented as a dialogue move of the form `alarm_triggered(T)`. The most common type of dialogue action is a term of the form `say(U)`, representing a request to speak an utterance abstractly represented by the term `U`. Other types of dialogue actions include modifying the display, changing the volume, and so on.

The information state is a vector, which in the current version of the system contains 26 elements. Some of these elements represent properties of the dialogue itself. In particular, the `last_state` element is a back-pointer to the preceding dialogue state, and the `expectations` element encodes information about how the next dialogue move is to be interpreted. For example, if a yes/no question has just been asked, the `expectations` element will contain information determining the intended interpretation of the dialogue moves `yes` and `no`.

The novel aspect of the Clarissa DM is that all *task* information is also uniformly represented as part of the information state. Thus for example the `current_location` element holds the procedure step currently being executed, the `current_alarms` element lists the set of alarms currently set, associating each alarm with a time and a message, and the `current_volume` element represents the output volume, expressed as a percentage of its maximum value. Putting the task information into the information state has the desirable consequence that actions whose effects can be defined in terms of their effect on the information state need not be specified directly. For example, the update rule for the dialogue move `go_to(Loc)` specifies among other things that the value of `current_location` element in the output dialogue state will be `Loc`. The specific rule does not also need to say that an action needs to be produced to update the GUI by scrolling to the next

location; that can be left to a general rule, which relates a change in the `current_location` to a scrolling action.

More formally, what we are doing here is dividing the work performed by the update function f into two functions, g and h. g is of the same form as f, i.e.

$$g : State \times Move \rightarrow State \times Actions$$

As before, this maps the input state and the dialogue move into an output state and a set of actions; the difference is that this set now only includes the *irreversible* actions. The remaining work is done by a second function

$$h : State \times State \rightarrow Actions$$

which maps the input state S and output state S' into the set of reversible actions required to transform S into S'; the full set of output actions is the union of the reversible and the irreversible actions. The relationship between the functions f, g and h can be expressed as follows. Let S be the input state, and M the input dialogue move. Then if $g(S, M) = \langle S', A_1 \rangle$, and $h(S, S') = A_2$, we define $f(S, M)$ to be $\langle S', o(A_1 \cup A_2) \rangle$, where o is a function that maps a set of actions into an ordered sequence.

In the Clarissa system, h is implemented concretely as the set of all solutions to a Prolog predicate which contains one clause for each type of difference between states which can lead to an action. Thus we have for example a clause which says that a difference in the `current_volume` elements between the input state and the output state requires a dialogue action that sets the volume; another clause which says that an alarm time present in the `current_alarms` element of the input state but absent in the `current_alarms` element requires a dialogue action which cancels an alarm; and so on. The ordering function o is defined by a table which associates each type of dialogue action with a priority; actions are ordered by priority, with the function calls arising from the higher-priority items being executed first. Thus for example the priority table defines a `load_procedure` action as being of higher priority than a `scroll` action, capturing the requirement that the system needs to load a procedure into the GUI before it can scroll to its first step.

6.2 Specific issues

6.2.1 "Undo" and "correction" moves

As already noted, one of the key requirements for Clarissa is an ability to handle "undo" and "correction" dialogue moves. The conventional approach, as for example implemented in the CommandTalk system [Stent et al., 1999], involves keeping a "trail" of actions, together with a table of inverses which allow each action to be undone. The extended information state approach described above permits a more elegant solution to this problem, in which corrections are implemented using the g and h functions together with the `last_state` element of the information state. Thus if we write u for the "undo" move, and $l(S)$ to denote the state that S's `last_state` element points to, we can define $g(S, u)$ to be $\langle l(S), \emptyset \rangle$, and hence $f(S, u)$ will be $\langle l(S), o(h(S, l(S))) \rangle$. Similarly, if we write $c(M)$ for the move which consists of a correction followed by M, we can define $f(S, c(M))$ to be $\langle S', o(A \cup h(S, S')) \rangle$, where S' and A are defined by $g(l(S), M) = \langle S', A \rangle$.

In practical terms, there are two main payoffs to this approach. First, code for supporting undos and corrections shrinks to a few lines, and becomes trivial to maintain. Second, corrections are in general faster to execute than they would be in the conventional approach, since the h

function directly computes the actions required to move from S to S', rather than first undoing the actions leading from $l(S)$ to S, and then redoing the actions from $l(S)$ to S'. When actions involve non-trivial redrawing on the visual display, this difference can be quite significant.

6.2.2 Confirmations

Confirmations are in a sense complementary to corrections. Rather than making it easy for the user to undo an action they have already carried out, the intent is to repeat back to them the dialogue move they appear to have made, and give them the option of not performing it at all. Confirmations can also be carried out at different levels. The simplest kind of confirmation echoes the exact words the system believed it recognised. It is usually, however, more useful to perform confirmations at a level which involves further processing of the input. This allows the user to base their decision about whether to proceed not merely on the words the system believed it heard, but also on the actions it proposes to take in response.

The information state framework also makes possible a simple approach to confirmations. Here, the key idea is to compare the current state with the state that would arise after responding to the proposed move, and repeat back a description of the difference between the two states to the user. To write this symbolically, we start by introducing a new function $d(S, S')$, which denotes a speech action describing the difference between S and S', and write the dialogue moves representing "yes" and "no" as y and n respectively. We can then define $f_{conf}(S, M)$, a version of $f(S, M)$ which performs confirmations, as follows. Suppose we have $f(S, M) = \langle S', A\rangle$. We define $f_{conf}(S, M)$ to be $\langle S_{conf}, d(S, S')\rangle$, where S_{conf} is constructed so that $f(S_{conf}, y) = \langle S', A\rangle$ and $f(S_{conf}, n) = \langle S, \emptyset\rangle$. In other words, S_{conf} is by construction a state where a "yes" will have the same effect as M would have had on S if the DM had proceeded directly without asking for a confirmation, and where a "no" will leave the DM in the same state as it was before receiving M.

There are two points worth noting here. First, it is easy to define the function f_{conf} precisely because f is side-effect free; this lets us derive and reason about the *hypothetical* state S' without performing any external actions. Second, the function $d(S, S')$ will in general be tailored to the requirements of the task, and will describe *relevant* differences between S and S'. In Clarissa, where the critical issue is which procedure steps have been completed, $d(S, S')$ describes the difference between S and S' in these terms, for example saying that one more step has been completed, or three steps skipped.

6.2.3 Querying the environment

An obvious problem for any side-effect free dialogue management approach arises from the issue of querying the environment. If the DM needs to acquire external information to complete a response, it may seem that the relationship between inputs and output can no longer be specified as a self-contained function.

The framework can, however, be kept declarative by splitting up the DM's response into two turns. Suppose that the DM needs to read a data file in order to respond to the user's query. The first turn responds to the user query by producing an action request to read the file and report back to the DM, and an output information state in which the DM is waiting for a dialogue move reporting the contents of the file; the second turn responds to the file-contents reporting action by using the new information to reply to the user. The actual side-effect of reading the file occurs

outside the DM, in the space between the end of the first turn and the start of the second. Variants of this scheme can be applied to other cases in which the DM needs to acquire external information.

6.2.4 Regression testing and evaluation

Regression testing and evaluation on context-dependent dialogue systems is a notoriously messy task. The problem is that it is difficult to assemble a reliable test library, since the response to each individual utterance is in general dependent on the context produced by the preceding utterances. If an utterance early in the sequence produces an unexpected result, it is usually impossible to know whether results for subsequent utterances are meaningful.

In our framework, regression testing of contextually dependent dialogue turns is unproblematic, since the input and output contexts are well-defined objects. We have been able to construct substantial libraries of test examples, where each example consists of a 4-tuple ⟨InState, DialogueMove, OutState, Actions⟩. These libraries remain stable over most system changes, except for occasional non-downward-compatible redesigns of the dialogue context format, and have proved very useful.

7 Summary and conclusions

We have presented a detailed description of a non-trivial spoken dialogue system, which carries out a potentially useful task in a very challenging environment. In the course of the project, we have addressed several general problems and developed what we feel are interesting and novel solutions. We conclude by summarising our main results.

7.1 Procedures

The problem of converting written procedures into voice-navigable documents is not as simple as it looks. A substantial amount of new information needs to be added, in order to make explicit the instructions which were implicit in the original document and left to the user's intelligence. This involves adding explicit state to the reading process. Jumps in the procedure can leave the state inconsistent, and it is not trivial to ensure that the system will always be able to recover gracefully from these situations.

7.2 Recognition

For this kind of task, there is reasonable evidence that grammar-based recognition methods work better than statistical ones. The extra robustness of statistical methods does not appear to outweigh the fact that the grammar-based approach permits tighter integration of recognition and semantic interpretation. Speech understanding performance was very substantially better with the grammar-based method. We were able to make a clear comparison between the two methods because we used a carefully constructed methodology which built the grammar-based recogniser from a corpus, but there is no reason to believe that other ways of building the grammar-based recogniser would have led to inferior results.

7.3 Response filtering

The SVM based method for performing response filtering that we have developed is considerably better than the naive threshold based method. It is also completely domain-independent, and offers several possibilities for further performance gains. We consider this to be one of the most significant research contributions made by the project.

7.4 Dialogue management

The fully declarative dialogue management framework that we have implemented is applicable to any domain, like ours, where the dialogue state can be fully specified. If this condition is met, our method is simple to implement, and offers a clean and robust treatment of correction and confirmation moves. Perhaps even more significantly, it also permits systematic regression testing of the dialogue management component as an isolated module.

7.5 General

The challenge of creating a dialogue system to support procedure execution on the ISS posed a number of interesting research problems. It also demanded production of a prototype system robust enough to pass the rigorous software approval process required for deployment in space. We have now reached a point in the project where we think we can claim that the research problems have been solved. The system as a whole performs solidly, and people both inside and outside NASA generally experience it as a mature piece of production-level software. This includes several astronauts with first-hand experience of using other procedure viewers during space missions.

As described earlier, we have been able to carry out detailed testing of the individual system components. We would now like to address the bottom-line question, namely whether Clarissa actually is an ergonomic win compared to a conventional viewer. Obviously, hands- and eyes-free operation is an advantage. This, however, must be balanced against the fact that reading from a screen is faster than reading aloud.

Based on initial studies we have carried out within the development group, our impression is that tight usability experiments are not straightforward to design; an astronaut who is using a speech-enabled procedure reader has to make non-trivial changes in his normal work practices, in order to fully exploit the new technology. In other words, the challenge is now to learn how to work efficiently together with a voice-enabled automatic assistant. We are currently discussing these issues with training and work practice groups at NASA, and hope to be able to report on them in a later paper.

References

[Allen et al., 2000] Allen, J., Byron, D., Dzikovska, M., Ferguson, G., Galescu, L., and Stent, A. (2000). An architecture for a generic dialogue shell. *Natural Language Engineering, Special Issue on Best Practice in Spoken Language Dialogue Systems Engineering*, pages 1–16.

[Bennett, 2003] Bennett, P. (2003). Using asymmetric distributions to improve text classifier probability estimates. In *Proceedings of the 26th ACM SIGIR Conference*, Toronto, Ontario.

[Carter, 2000] Carter, D. (2000). Choosing between interpretations. In Rayner, M., Carter, D., Bouillon, P., Digalakis, V., and Wirén, M., editors, *The Spoken Language Translator*. Cambridge University Press.

[Cortes et al., 2004] Cortes, C., Haffner, P., and Mohri, M. (2004). Rational kernels: Theory and algorithms. *Journal of Machine Learning Research*, pages 1035–1062.

[Dowding and Hieronymus, 2003] Dowding, J. and Hieronymus, J. (2003). A spoken dialogue interface to a geologist's field assistant. In *Proceedings of HLT-NAACL*, Edmonton, Alberta.

[Duda et al., 2000] Duda, R., Hart, P., and Stork, H. (2000). *Pattern Classification*. Wiley, New York.

[Haffner et al., 2003] Haffner, P., Cortes, C., and Mohri, M. (2003). Lattice kernels for spoken-dialog classification. In *Proceedings of ICASSP 2003*, Hong Kong.

[Joachims, 1998] Joachims, T. (1998). Text categorization with support vector machines: Learning with many relevant features. In *Proceedings of the 10th European Conference on Machine Learning*, Chemnitz, Germany.

[Joachims, 2005] Joachims, T. (2005). http://svmlight.joachims.org/. As of 15 March 2005.

[Knight et al., 2001] Knight, S., Gorrell, G., Rayner, M., Milward, D., Koeling, R., and Lewin, I. (2001). Comparing grammar-based and robust approaches to speech understanding: a case study. In *Proceedings of Eurospeech 2001*, pages 1779–1782, Aalborg, Denmark.

[Larsson and Traum, 2000] Larsson, S. and Traum, D. (2000). Information state and dialogue management in the TRINDI dialogue move engine toolkit. *Natural Language Engineering, Special Issue on Best Practice in Spoken Language Dialogue Systems Engineering*, pages 323–340.

[Lodhi et al., 2002] Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., and Watkins, C. (2002). Text classification using string kernels. *Journal of Machine Learning Research*, pages 419–444.

[Martin et al., 1998] Martin, D., Cheyer, A., and Moran, D. (1998). Building distributed software systems with the open agent architecture. In *Proceedings of the Third International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, Blackpool, Lancashire, UK.

[Navia-Vázquez et al., 2004] Navia-Vázquez, A., Pérez-Cruz, F., Artés-Rodrguez, A., and Figueiras-Vidal, A. R. (2004). Advantages of unbiased support vector classifiers for data mining applications. *Journal of VLSI Signal Processing Systems*, 37(1-2):1035–1062.

[Nuance, 2003] Nuance (2003). http://www.nuance.com. As of 25 February 2003.

[Pulman, 1992] Pulman, S. (1992). Syntactic and semantic processing. In Alshawi, H., editor, *The Core Language Engine*, pages 129–148. MIT Press, Cambridge, Massachusetts.

[Rayner, 1988] Rayner, M. (1988). Applying explanation-based generalization to natural-language processing. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1267–1274, Tokyo, Japan.

[Rayner and Hockey, 2003] Rayner, M. and Hockey, B. (2003). Transparent combination of rule-based and data-driven approaches in a speech understanding architecture. In *Proceedings of the 10th Conference of the European Chapter of the Association for Computational Linguistics*, Budapest, Hungary.

[Rayner et al., 2002] Rayner, M., Hockey, B., and Dowding, J. (2002). Grammar specialisation meets language modelling. In *Proceedings of the 7th International Conference on Spoken Language Processing (ICSLP)*, Denver, CO.

[Rayner et al., 2003] Rayner, M., Hockey, B., and Dowding, J. (2003). An open source environment for compiling typed unification grammars into speech recognisers. In *Proceedings of the 10th Conference of the European Chapter of the Association for Computational Linguistics (interactive poster and demo track)*, Budapest, Hungary.

[Regulus, 2005] Regulus (2005). http://sourceforge.net/projects/regulus/. As of 8 January 2005.

[Shawe-Taylor and Cristianini, 2004] Shawe-Taylor, J. and Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*. Cambridge University Press.

[Stent et al., 1999] Stent, A., Dowding, J., Gawron, J., Bratt, E., and Moore, R. (1999). The CommandTalk spoken dialogue system. In *Proceedings of the Thirty-Seventh Annual Meeting of the Association for Computational Linguistics*, pages 183–190.

[van Eijck and Moore, 1992] van Eijck, J. and Moore, R. (1992). Semantic rules for English. In Alshawi, H., editor, *The Core Language Engine*, pages 83–116. MIT Press.

[van Harmelen and Bundy, 1988] van Harmelen, T. and Bundy, A. (1988). Explanation-based generalization = partial evaluation (research note). *Artificial Intelligence*, 36:401–412.

[Yarowsky, 1994] Yarowsky, D. (1994). Decision lists for lexical ambiguity resolution. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, pages 88–95, Las Cruces, New Mexico.

[Zadrozny and Elkan, 2002] Zadrozny, B. and Elkan, C. (2002). Transforming classifier scores into accurate multiclass probability estimates. In *Proceedings of the 8th ACM SIGKDD Conference*, Edmonton, Alberta.