Basurde[lite]
A human-machine dialogue system for accessing railway timetable information

Roger Trias-Sanz
Master’s Thesis advisor: José B. Mariño Acebal

September 2001
The CD-ROM contains the full version of this document (including the commented source files) in DVI, PostScript and PDF formats. It may contain the \LaTeX\ sources for this document, the source files for the pictures (in \texttt{xfig} format) and the full sources for Basurde[lite], in various programming languages using the \texttt{noweb} literate programming tool.

\texttt{BibLaTeX} document entry:

\begin{verbatim}
\@MastersThesis{TriasSanz:Basurde,
  author = {Trias-Sanz, Roger},
  title = {Basurde[lite]. \{A\} human-machine dialogue system for accessing railway timetable information},
  school = {Escola T(e)nica Superior d'Enginyeria de Telecomunicaci\`o de Barcelona, Universitat Polit\`ecnica de Catalunya},
  year = 2001,
  month = sep,
  address = {Jordi Girona, 1-3; E-08034 Barcelona; Spain}
}
\end{verbatim}
Agraïments

a en JoseMariño, el director del PFC, i
a l’infatigable Antonio Bonafonte, sense els
quals aquest treball no hauria estat possible;
al meu pare i la meva mare;
a la Marta i colla, la Young i colla, i la resta dels
meus amics, per ser-hi;
i a tothom qui m’ha ajudat a fer les proves del
sistema.

if you ever get close to a human
and human behaviour
be ready to get confused
there’s definitely no logic
to human behaviour
but yet so irresistible
there’s no map
to human behaviour

they’re terribly moody
then all of a sudden turn happy
but, oh, to get invoked in the exchange
of human emotions is ever so satisfying

there’s no map
and a compass
wouldn’t help at all
human behaviour

—Björk and N. Hooper. “Human Behaviour”
Contents

I Developing the system 1

1 Introduction 3

1.1 Human-machine dialogue systems 3
1.2 The Basurde project 4
1.3 The Basurde[lite] project 5
1.4 Organization of this document 7
1.5 Notation 8

1.5.1 Backus-Naur form (BNF) 8

2 Structure of the Basurde[lite] system 11

2.1 Modules 12

2.1.1 Functions of the Basurde modules 14
2.1.2 Mixed C++/Tcl programming 15
2.1.3 Structure of one Basurde[lite] module 17

2.2 Module interfaces 18

2.2.1 Protocol stack structure 18
2.2.2 Frames 19
2.2.3 Changes to the Basurde communications protocol 22
2.2.4 Module invocation interface 23

3 Designing the dialogue application 25

3.1 Overall dialogue strategy 25

3.1.1 Dialogue goals 25
3.1.2 System ergonomics and usability 28
3.1.3 Confirmation strategy 29
3.1.4 Technical constraints 30
3.1.5 Dialogue graphs ........................................... 31
3.1.6 Design method ........................................... 38
3.2 Output sentences ........................................... 39
  3.2.1 Date and time expressions .......................... 43
3.3 Input sentences ........................................... 44
  3.3.1 Recognizer states .................................... 45
  3.3.2 Recognized sentences ................................. 45
  3.3.3 Date and time expressions .......................... 51

4 The Basurde[lite] modules ..................................... 55
  4.1 Speech recognition ...................................... 55
  4.2 Natural language understanding ........................ 56
    4.2.1 Proposed solutions ................................ 57
    4.2.2 The choice for Basurde[lite] ...................... 58
  4.3 Dialogue manager ........................................ 65
    4.3.1 Proposed solutions ................................ 65
    4.3.2 The choice for Basurde[lite] ...................... 67
    4.3.3 An example ......................................... 74
  4.4 Oral response generator ................................ 77

5 Implementing the dialogue application ......................... 81
  5.1 Language models for the speech recognizer .............. 81
    5.1.1 Layers ............................................. 83
    5.1.2 Comments on the trash models .................... 85
  5.2 Frames on the input side ................................ 86
    5.2.1 Comprehension failures (user made) .............. 86
    5.2.2 System failures .................................... 86
    5.2.3 Giving information ................................ 87
    5.2.4 Confirming system beliefs ........................ 89
  5.3 Frames on the output side ................................ 90
    5.3.1 Comprehension failures ............................ 90
    5.3.2 Offers and politeness .............................. 91
    5.3.3 Questions .......................................... 92
    5.3.4 Giving information ................................ 94
CONTENTS

5.3.5 Explicitly requesting user confirmation ............... 96
5.3.6 Implicitly requesting user confirmation ............... 98
5.4 Frame types, case names and case values ............... 100
  5.4.1 Accents and capital letters .......................... 100
  5.4.2 Frame types ........................................ 101
  5.4.3 Case names and values ................................ 101
  5.4.4 Case values for single dates ......................... 106
  5.4.5 Case values for date intervals ....................... 108
  5.4.6 Case values for single times ......................... 110
  5.4.7 Case values for time intervals ....................... 111
  5.4.8 Recognition states for the NLU module ............. 112
5.5 Dialogue manager script files ............................ 113
  5.5.1 Node class hierarchy ................................ 113
  5.5.2 Frame acceptors ..................................... 118
5.6 Multiple languages ........................................ 118

II Results .................................................. 123

6 Evaluation ............................................... 125
  6.1 Description of measured quality parameters ............ 126
  6.2 Call scenarios .......................................... 128
  6.3 Results ................................................. 129
    6.3.1 Qualitative remarks ................................ 129
    6.3.2 Quantitative results ................................ 130
    6.3.3 User comments ...................................... 140
    6.3.4 Comparison to other systems ....................... 140
  6.4 Two example dialogues .................................. 142

7 Conclusion ............................................... 147
  7.1 Concluding remarks .................................... 147
  7.2 Lessons learned ....................................... 148
  7.3 Future work ........................................... 149
III Appendices

A Guide for running Basurde[lite]
A.1 Directory structure ............................................. 153
A.2 Command-line interface ....................................... 156
  A.2.1 Basurde[lite] modules .................................... 156
  A.2.2 Audio server .............................................. 157
  A.2.3 Speech recognizer ......................................... 157
  A.2.4 Speech synthesizer ....................................... 158
  A.2.5 Communications manager ................................. 158
A.3 Starting it all on one computer ................................ 158
  A.3.1 The TkBasurde program .................................. 158
  A.3.2 The sa-runner program ................................... 159
A.4 The top-level makefile ......................................... 159
A.5 Logging ............................................................ 159
  A.5.1 Tcl logging routines ...................................... 160
  A.5.2 Messages on standard output and standard error ...... 160
  A.5.3 Call codes ................................................ 160
  A.5.4 Call recording ............................................ 160
  A.5.5 Other logging ............................................. 161
A.6 Modified Basurde modules ..................................... 161

B Guide for the application developer ............................. 163
B.1 What to change .................................................. 163
  B.1.1 Application definition files .............................. 163
  B.1.2 Language model for the speech recognizer .............. 164
  B.1.3 Tcl communication loop .................................. 164
B.2 How to modify source files ..................................... 165
B.3 Tcl APIs for the common helper routines ....................... 166
  B.3.1 Tcl format for dates ...................................... 166
  B.3.2 Tcl format for times ..................................... 167
  B.3.3 Commands for date manipulation ........................ 167
  B.3.4 Commands for time manipulation ........................ 168
B.4 Tcl APIs for the NLU module .................................. 168
CONTENTS

B.4.1 Tcl representation for a recognized sentence ..... 168
B.4.2 Manipulating the current state .................. 169
B.4.3 Working with yield strings ...................... 170
B.4.4 Tags, groups and goals .......................... 171
B.4.5 Other commands related to translation ............. 178
B.4.6 Tcl commands for handling inter-module communication 179
B.5 Tcl APIs for the dialogue manager module .......... 181
B.5.1 Node paths .................................... 181
B.5.2 Commands for creating new nodes ................. 182
B.5.3 Outer-level dialogue control ...................... 183
B.5.4 Accessing the value slot and node variables ......... 186
B.5.5 Controlling how to ask questions about a node .... 187
B.5.6 Setting the acceptor function .................... 188
B.5.7 Setting the accessor function ..................... 189
B.5.8 Setting the value-changing function ............... 191
B.5.9 Managing the agenda ........................... 191
B.5.10 Other commands ................................. 192
B.5.11 Tcl commands for handling inter-module communication 194
B.6 Tcl APIs for the ORG module .......................... 196
B.6.1 Frame description lists .......................... 196
B.6.2 Tcl commands for describing translation rules ........ 197
B.6.3 Other Tcl commands related to translation .......... 198
B.6.4 Tcl commands for handling inter-module communication 198
B.7 How to test the application definition files .......... 200
B.7.1 Natural language understanding module ............ 200
B.7.2 Dialogue manager ................................ 201
B.7.3 Oral response generator .......................... 203
B.8 A brief Tcl tutorial ................................ 203
B.8.1 Statements and grouping .......................... 203
B.8.2 Variables and substitution ....................... 204
B.8.3 Quotes, quoting with curly brackets, lists .......... 204
B.8.4 User-defined procedures, control structures, grouping .. 205

C The commented source files .......................... 207
CONTENTS

C.1 Grammar for the speech recognizer .............................. 207
  C.1.1 The gram-bnf*.txt file .................................. 207
  C.1.2 The word-expand.tcl file ................................. 228
  C.1.3 The speech recognizer file basurde-lite.arq ............... 229
  C.1.4 The exceptions-es.txt file .............................. 233
  C.1.5 The exceptions-nes.txt file ............................. 234
  C.1.6 The processa-bnf.sh file ................................. 234
  C.1.7 The fill-in-exceptions.tcl file .......................... 242
  C.1.8 The dict-model-exc.tcl file ............................. 243
  C.1.9 The dict-model-exc.txt file .............................. 244
  C.1.10 The gen-triv-gram-bnf.tcl file ......................... 245

C.2 Common helper routines .................................... 245
  C.2.1 The ParseString C++ files ............................... 245
  C.2.2 The date-time C++ files ................................. 250
  C.2.3 The utils C++ files ...................................... 262
  C.2.4 The argminmax C++ files ................................ 267
  C.2.5 The logging.tcl file .................................... 269

C.3 Natural language understanding ............................ 270
  C.3.1 The compren C++ file .................................. 270
  C.3.2 The com-comm-tcl C++ files ............................. 280
  C.3.3 The com-tcl C++ files .................................. 290
  C.3.4 The basic-types C++ header file ......................... 305
  C.3.5 The Tags C++ files .................................... 306
  C.3.6 The TaggedString C++ files ............................. 308
  C.3.7 The Parse C++ files .................................... 312
  C.3.8 The com-config C++ files ............................... 327
  C.3.9 The com-parse C++ files ................................ 330
  C.3.10 The compren.tcl file .................................. 347
  C.3.11 The comlib.tcl file ................................... 360
  C.3.12 The numbers.tcl file .................................. 370
  C.3.13 The dates.tcl file .................................... 372
  C.3.14 The date-goals.tcl file ................................ 378
  C.3.15 The times.tcl file .................................... 382
  C.3.16 The time-goals.tcl file ............................... 388
C.3.17 The places.txt file ........................... 394
C.3.18 The train-types.txt file ...................... 398

C.4 Dialogue manager ................................ 399
  C.4.1 The dialog C++ files ......................... 399
  C.4.2 The tcl-comm C++ files ...................... 409
  C.4.3 The Agenda C++ files ........................ 420
  C.4.4 The Env C++ files ............................ 423
  C.4.5 The Frames C++ files ......................... 426
  C.4.6 The ValueSlot C++ files ..................... 432
  C.4.7 The KNode C++ files .......................... 442
  C.4.8 The Manager C++ files ....................... 473
  C.4.9 The STLOutput C++ file ...................... 507
  C.4.10 The debug C++ file .......................... 508
  C.4.11 The dia-tcl C++ files ....................... 509
  C.4.12 The tcl-innode C++ files .................... 523
  C.4.13 The dialog.tcl file .......................... 538
  C.4.14 The dialib.tcl file ........................... 555
  C.4.15 The output.tcl file ........................... 570
  C.4.16 The reduce.tcl file ........................... 576
  C.4.17 The Date.tcl file ............................ 586
  C.4.18 The Place.tcl file ............................ 592
  C.4.19 The Time.tcl file ............................. 595
  C.4.20 The Train.tcl file ............................ 619
  C.4.21 The TrainType.tcl file ....................... 620
  C.4.22 The TrainService.tcl file ................... 621
  C.4.23 The HLDate.tcl file .......................... 622
  C.4.24 The HLTime.tcl file ........................... 625
  C.4.25 The Boolean.tcl file ......................... 629
  C.4.26 The GenSym.tcl file ........................... 630
  C.4.27 The Post.tcl file .............................. 631
  C.4.28 The MultipleChoice.tcl file ................. 633
  C.4.29 The Selllista.tcl file ....................... 634
  C.4.30 The dbase.tcl file ............................ 635
  C.4.31 The cgi-class.txt file ....................... 646
C.4.32 The cgi-places.txt file ........................................ 647
C.4.33 The cgi-serv.txt file ........................................... 651
C.4.34 The cgi-traintype.txt file .................................... 651
C.4.35 The test.tcl file ................................................ 652
C.5 Oral response generator ............................................. 655
C.5.1 The resporal C++ file ........................................... 655
C.5.2 The res-comm.tcl C++ files ................................. 665
C.5.3 The res-config C++ files .................................... 673
C.5.4 The res-tcl C++ files ....................................... 682
C.5.5 The res-xlat C++ files .................................... 691
C.5.6 The dates.tcl file ........................................... 695
C.5.7 The numbers.tcl file ....................................... 698
C.5.8 The times.tcl file ........................................... 703
C.5.9 The train-types.txt file .................................. 708
C.5.10 The places.txt file ...................................... 709
C.5.11 The resplib.tcl file .................................... 713
C.5.12 The resporal.tcl file .................................. 720
C.6 The makefiles ....................................................... 739
C.6.1 ./Makefile ..................................................... 739
C.6.2 ./Makefile.defs ............................................. 743
C.6.3 ./gram-recon/Makefile ................................ 744
C.6.4 ./common/Makefile ....................................... 745
C.6.5 ./comprensio/Makefile ................................ 746
C.6.6 ./dialeg/Makefile ......................................... 748
C.6.7 ./resporal/Makefile .................................... 750

Program index .................................................................. 753
List of Tables

3.1 Recognizer states ............................................ 45

5.1 Language codes used in Basurde[lite] ...................... 121

6.1 Comparison to other systems ................................. 141

C.1 Top-level recognizer grammars ............................... 208
C.2 Top-level additional non-terminals for recognizer grammars . 208
C.3 Second-level recognizer grammars ............................ 209
LIST OF TABLES
List of Figures

2.1 Network connections between Basurde[lite] modules . . . . . . . 12
2.2 Data flows between Basurde[lite] modules . . . . . . . . . . . . . 13
2.3 Structure of a generic Basurde[lite] module . . . . . . . . . . . . 17
2.4 Communication interfaces and layers . . . . . . . . . . . . . . . 18

3.1 General dialogue overview . . . . . . . . . . . . . . . . . . . . . . 33
3.2 Top-level dialogue automaton . . . . . . . . . . . . . . . . . . . . 34
3.3 Handling short lists . . . . . . . . . . . . . . . . . . . . . . . . . . 35
3.4 Handling long lists . . . . . . . . . . . . . . . . . . . . . . . . . . 36
3.5 Handling a single train . . . . . . . . . . . . . . . . . . . . . . . . 37
3.6 Offer to look for other trains . . . . . . . . . . . . . . . . . . . . . 38

4.1 Converting word sequences to frames . . . . . . . . . . . . . . . . 56
4.2 Part of an example parsing of an arrival time . . . . . . . . . . . 59
4.3 An example semi-deterministic parsing of an arrival time . . . 61
4.4 How the NLU parser works . . . . . . . . . . . . . . . . . . . . . . 62
4.5 Temporarily storing structural information about the parsed string 64
4.6 A task tree . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
4.7 Confirmation status after a dialogue fragment . . . . . . . . . . . 69
4.8 Dialogue manager example 1/5 . . . . . . . . . . . . . . . . . . . 74
4.9 Dialogue manager example 2/5 . . . . . . . . . . . . . . . . . . . 75
4.10 Dialogue manager example 3/5 . . . . . . . . . . . . . . . . . . . 76
4.11 Dialogue manager example 4/5 . . . . . . . . . . . . . . . . . . . 76
4.12 Dialogue manager example 5/5 . . . . . . . . . . . . . . . . . . . 77
4.13 An example of ORG frame matching and translation . . . . . . 78

5.1 Layered structure of the language model . . . . . . . . . . . . . . 83
LIST OF FIGURES

5.2 Node classes: the Application class ........................................ 114
5.3 Node classes: the Place class ................................................. 116
5.4 Node classes: the Date class .................................................. 116
5.5 Node classes: the Time class .................................................. 117
5.6 Frame acceptors: the top-level .............................................. 119
5.7 Frame acceptors: dates ........................................................... 120
5.8 Frame acceptors: times ........................................................... 120
5.9 Frame acceptors: places .......................................................... 121

6.1 Results from all tests ............................................................. 132
6.2 Effect of user experience on speech technologies ......................... 134
6.3 Effect of user experience with the system .................................... 135
6.4 Effect of scenario difficulty ...................................................... 137
6.5 Effect of ambient noise ............................................................ 138
6.6 Correlation of results to dialogue success .................................... 139

A.1 Directory structure ............................................................... 154

B.1 How to use other communication protocols ................................ 165
B.2 Relative paths ........................................................................... 182

C.1 Dialogue automaton for single times ......................................... 597
C.2 Dialogue automaton for time intervals ....................................... 605
Abstract

This Master's Thesis is the design and implementation of a human-machine dialogue system for accessing railway timetable information. The dialogue is machine-driven and uses a limited vocabulary and grammar. The system attempts to be similar, in functionality and interfaces, to the Basurde system, which is human-driven and gives the user more freedom of vocabulary and grammar.

Reducing the accepted language can make the dialogues less fluent and natural, but at the same time can improve the success rate of the speech recognizer. The goal of this project is to develop a dialogue system similar to Basurde in much less time, and to explore whether this reduction actually causes an improvement in the overall dialogue quality.
Resum

Aquest Projecte de Final de Carrera consisteix en el disseny i implementació d’un sistema de diàleg persona-màquina per a l’accés a informació d’horaris de trens. El diàleg és guiat per la màquina i utilitza un vocabulari i gramàtica límitats. El sistema intenta proporcionar una funcionalitat i interfícies semblants a les del sistema Basurde, un sistema de diàleg molt més complex, guiat per l’usuari, i amb un vocabulari i gramàtica més amplis.

Simplificar el llenguatge acceptat pot fer els diàlegs més artificials i menys fluïds, però alhora pot reduir la taxa d’error del reconeixedor de veu. L’objectiu d’aquest projecte és desenvolupar un sistema de diàleg semblant al Basurde en molt menys temps, i explorar si aquesta simplificació produeix una millora en la qualitat global del diàleg.
xxii
Part I

Developing the system
Chapter 1

Introduction

This document describes the Basurde[lite] system, a human-machine natural language dialogue system for accessing railway timetable information by telephone.

1.1 Human-machine dialogue systems

Human-machine dialogue systems are computer systems which interact with the human user by means of conversations or dialogues in a human language (not a computer language).

We are used now to graphical user interfaces, where the user communicates with the computer by manipulating some graphical elements on a screen (scroll bars, buttons, check-boxes, menus, ...), and the computer communicates with the user by showing information on a computer screen. Textual user interfaces, where the keyboard is the main communicating element, are also common. Conversational user interfaces are not so widely used as textual or graphical ones, but have some very interesting characteristics:

1. They require no, or very little, interface equipment. For example, the Basurde[lite] system can be accessed from any telephone.

2. Interaction is easier for the user since her natural language is used (though there is some discussion here: for some tasks using an artificial language might indeed be more “natural”).

3. Little or no user training is required.

4. Blind people and people with certain motion handicaps can use these systems.

5. Conversational user interfaces can be used on a wider range of situations than graphical or textual user interfaces.
On the other hand, there are also some drawbacks associated with human-machine dialogue systems:

1. Some information is better transmitted to the user in visual form. Voice and conversation is too linear and slow, and the human short-term memory is too limited for some large pieces of data. Some systems solve this by being multi-modal, combining conversational, visual and other kinds of interface.

2. Current systems are far from perfect. The computer does not always correctly understand what the user said, and knows only about a limited set of concepts. Dialogues, then, are not as natural as human-to-human dialogues.

What is more, it seems that more natural dialogues require larger and more complex input grammars (models of the user’s language). However, as larger input grammars are used, worse performances are obtained from speech recognizers (the part which converts spoken input into text form). It seems then that there is a tradeoff between language flexibility and dialogue naturalness, on one side, and recognition and dialogue success rates, on the other.

Since these are computer interfaces to people, not other computers, human factors have to be taken into account when designing a conversational interface. A dialogue system’s success greatly depends on ergonomics and usability factors. See [17] for a discussion of human factors applied to conversational interfaces. For more information on dialogue systems, refer to [48], an informal review of the issues related to conversational interfaces and human-machine dialogue systems, and to [39], a more detailed discussion.

1.2 The Basurde project

On October 1998, research teams from several universities (Euskal Herriko Unibertsitatea¹, Universitat Jaume I², Universitat Politecnica de Catalunya³ ⁴, Universitat Politecnica de Valencia⁵ and Universidad de Zaragoza⁶) started the Basurde joint project.

Its goal is to develop a human-machine dialogue system for accessing railway timetable information by telephone, in which the human user has the dialogue

¹http://sirius.ws.kc.ehu.es/~voz
²http://sic.ujie.es/ocit/grups/grup12.html
³http://wwwlsi.upc.es/~acquilex/nhr.html
⁴http://gps-ts.upc.es/veu
⁵http://www.upv.es/info/DSIC/index.html
⁶http://www.cps.unizar.es/deps/GTC/gtc.html
initiative, and where the computer can understand complex user utterances and has a wide vocabulary. People can call this computer system using a normal telephone, and talk with the computer using natural language in order to obtain information about railway schedules. The system is designed to work with the Spanish language.

The project was scheduled to end on September 2001.

The Basurde system will provide, if successful, a very interesting and powerful human-machine dialogue system, since at any dialogue turn the user will be able to say almost anything—related to train schedules—and be understood by the machine. If everything goes correctly, the dialogues will then be very natural and user satisfaction will be high.

However, Basurde is a very difficult project. We expect its development to take a long time. Also, since the computer must understand a large number of words and language constructs, it could be that the system has a high error rate, or works very slowly.

1.3 The Basurde[lite] project

That is where the Basurde[lite] project starts. It works in the same domain as Basurde (railway timetable information), and is prepared to give to the user the same kind of information as Basurde does. But it is very simplified, both in the accepted vocabulary, in the language constructs that the system understands, and in the dialogue events that it can handle. This simplification is the reason for the “lite” suffix in the name.

When several design options were possible, we chose the simplest valid option. The reason is that the complex option is already being used, in principle, for the full Basurde project, and the goal of the Basurde[lite] project is to improve transaction success, user satisfaction and development time from what the full Basurde provides, by sacrificing completeness and flexibility in favor of simplicity.

More precisely, the Basurde[lite] system is a machine-driven, limited-vocabulary and limited-grammar human-machine natural language dialogue system for accessing train timetable information by telephone.

The dialogues are machine-driven and the computer system understands a limited vocabulary and grammar. This means that the user is not free to ask any question she wants or change subject; instead, the computer takes control of the dialogue by asking the user specific questions. The user is expected to answer these questions and not to try to get the dialogue initiative. Also, the computer understands a very limited language: Only simple sentences, using words from a small predefined set, and using only some specific language constructs, will be understood by the computer. This does not mean that the user
must use over-simplified, unnatural speech, but rather that only the most basic meaning of sentences will be understood, and that very complex sentences, or simple sentences using uncommon words, may not be understood at all.

Some experiments ([9]) show that user satisfaction may actually be higher with machine-driven dialogues than with human-driven ones, mainly because the user’s expectations are much higher with human-driven systems, whereas the user clearly perceives machine-driven systems as computer-based, and accepts their lack of flexibility and naturalness.

It is expected that, because of these simplifications, the Basurde[lite] system will take significantly shorter to develop than its parent system, Basurde. We also expect that it will have a better recognition success rate and a higher transaction success rate (more dialogues will end with the user getting the desired information), despite the less natural interface.

On the other hand, the user will feel that she is interacting with a less intelligent agent than with Basurde. It is not clear if this is a disadvantage: Some authors suggest that it is not a good idea to model human-machine dialogues after human-human ones.

Some studies show that humans, when engaged in dialogues with machines that act mechanically in an obvious way, tend to adapt their communication to their preconceptions about machine behavior; this may make it easier for the machine to understand them, and increase dialogue success. In contrast, humans tend to have too high expectations of dialogue systems that act in a more human-like way; they use complex sentences that the machine cannot understand, which lowers the dialogue success rate.

Basurde[lite] is then a reduced-risk version of Basurde. The chances of the project succeeding are higher, but the resulting product will be less interesting.

Designing and implementing the Basurde[lite] system is the goal of this Master’s Thesis. The project includes an evaluation phase where objective and subjective data about the system’s performance is gathered. This will help us determine whether we actually have a better recognition performance, and whether the overall user satisfaction increases because of this better performance, or rather it decreases because of the less flexible dialogue patterns.

Being a derivation of Basurde, Basurde[lite] will reuse some of the work of its parent project. We use the same system architecture, which includes a particular division of work into several modules (see chapter 2), the definition of each module’s functionality, and an inter-module communication protocol. We will reuse, with minimal modifications, some of the Basurde modules. We will use the same packet format for inter-module communications, and we will reuse the concept, and overall format, of frames (see 2.2.2 on page 20). Our system will accept the same train stations, cities, train types, train services and
1.4. ORGANIZATION OF THIS DOCUMENT

ticket classes as the Basurde system.
Some of this reuse is interesting because it reduces the amount of work required for Basurde[lite]. In addition, since we use, at least partially, the same interfaces, Basurde[lite] modules may be used in the parent Basurde system. The oral response generator, which generates Spanish text given a description of the meaning of a sentence, requires little intelligence and is a good candidate for this.

1.4 Organization of this document

This chapter (chapter 1) gives a brief introduction into the subject of human-machine dialogue systems, presents the Basurde and the Basurde[lite] system, and gives the rationale behind Basurde[lite].

In chapter 2 we present the structure of the Basurde[lite] system. We describe the function of the individual components and the interfaces between these components.

The Basurde[lite] system is roughly divided in two parts: a set of programs, which we call modules, more or less independent of the actual dialogue application (for example, they could also be used for a hotel reservation dialogue system) and a set of files, which we call the application definition files, that describe the specific dialogue application (accessing train timetable information, in this case).

The design process for the dialogue application (which includes deciding the dialogue structure, the kind of questions that should be asked, what kind of information can be obtained from the system, . . . ) is described in chapter 3. Most readers should not need to read in much detail the long lists of expressions and sentences.

In chapter 4 we describe in detail each module of the Basurde[lite] system. We talk about their common structure. We list the several available options for designing each module, and comment the reasons that led us to each design and implementation choice.

In chapter 5 we explain in detail the implementation process for the application definition files, that is, how we get from the application design to a set of files that describe that application to the Basurde[lite] modules. The long lists in this chapter may be safely skimmed through.

The main part of this document ends with chapter 6, an evaluation of the technical and subjective performance of the system. We give the evaluation procedure and its results. Finally, we list some conclusions in chapter 7.

The appendices will be useful to a dialogue application developer that plans to use the Basurde[lite] system. A reader who is not interested in the detailed
implementation of the system should skip them.

In appendix A we describe the command-line interface to the Basurde[lite] modules, and give all the information that is needed to start a complete Basurde[lite] system on one or more computers. A system manager who is responsible for a Basurde[lite] system should read this chapter.

In appendix B we give hints on how to modify the Basurde[lite] system to implement a different dialogue application. In most cases this requires changing the application definition files only, so most of this chapter is devoted to this. A dialogue application developer that wants to use Basurde[lite] as a starting point for a different dialogue application should read this chapter and is also encouraged to skim through appendix C. Finally, appendix C contains the full commented source code for the modules, the application definition files, and several auxiliary files. It should only be read by the advanced developer, one who is looking for hints on solving an obscure problem, or a dialogue application developer that wants to use Basurde[lite] as a starting point for a very different dialogue application.

1.5 Notation

1.5.1 Backus-Naur form (BNF)

In some parts of this document we use grammars to define simple languages. We use Backus-Naur form (BNF) representations of regular expressions for this. Since some readers may not be familiar with BNF, and also since there are a lot of slightly different BNF notations around, we describe the notation used for this:

1. When explicitly given, the left-hand side of the rule will be written first, separated from the right-hand side by $\to$ or $::=$.

2. We use $[x]$ for optionality. This means that $x$ is optional at this point. For example, head[s] generates “head” and “heads”.

3. We use the Kleene star, $x*$, for zero or more repetitions of $x$. Also, $x+$ means $xx*$, one or more repetitions of $x$.

4. We use parentheses () for grouping.

5. We use $|$ for choice. For example, with number ::= 0|1|2|3, the non-terminal number generates “0”, “1”, “2” and “3”.

6. We use different fonts for terminals (words that cannot be further expanded according to a grammar) and non-terminals (words which are
also left-hand-sides of rules in the grammar). Which is which should normally be obvious from the context. We tend to use italic text for non-terminals and **boldface** or *typewritten* text for terminals. Do not mistake *, a Kleene star, for *, a terminal asterisk.
Chapter 2

Structure of the Basurde[lite] system

In this chapter we describe the structure of the Basurde[lite] system, and we talk briefly about its components and interfaces.

The implementation of the Basurde[lite] system consists of several programs, together with the configuration files that control their behavior. These programs communicate with each other using the Basurde communications protocol [30] (libcom), which allows modules to run across a network. This distributed approach was chosen for Basurde since it allowed teams in different locations to cooperate while each worked locally on its module.

Some of these modules have already been developed in the Basurde project, and have been reused for the Basurde[lite] module. The Basurde and Basurde[lite] systems are mostly compatible at the libcom level, which allows for some module reuse.

In figure 2.1 (page 12) we can see the modules and the network connections that get established between them.

The system uses a centralized approach where most data goes through a central module. If we ignore this, figure 2.2 (page 13) shows the data flows between modules. It also shows what the several configuration files act on.

In these figures, the light blue blocks are those developed as part of the Basurde[lite] project. Even if not an original goal for Basurde[lite], we also developed audio servers for ISDN modems and for sound cards, both running on Linux, reusing some parts of the original Basurde audio server.

The decoupling between programs and configuration files (application definition files) permits an easier adaptation of the system to a new application: In most cases, only the application definition files will need to be changed. The configuration files are actually Tcl programs [36], which gives extra flexibility
and power to the developer.

However, for applications which are not very similar to Basurde[lite], it is possible that the programs have to be modified too. They have been programmed so that these modifications are relatively easy to do.

Basurde[lite] is not a set of programs for developing all kinds of human-machine dialogue application. It is a set of programs for dialogue applications, general enough and yet designed with a specific application in mind, and a set of application definition files which can be understood by these programs, all designed to obtain a specific dialogue system: a restricted dialogue system for accessing timetable information for the Spanish state railway company, RENFE\(^1\).

2.1 Modules

In this section we briefly describe the functions of each Basurde (and Basurde[lite]) module, we talk about the dual-language nature of the Basurde[lite] modules, and we give their general common structure.

\(^1\)http://www.renfe.es
Figure 2.2: Data flows between Basurde[lite] modules

- Designed and implemented in the Basurde[lite] project
- Application definition files
2.1.1 Functions of the Basurde modules

The modules in the Basurde[lite] system are:

1. The *communications manager* acts as a communications hub: Following the Basurde communications protocol, almost all inter-module communication goes through the communications manager. Only audio samples moving to and from the audio server don’t. The communications manager also keeps a record of which modules are running, and the overall state of a call.

2. The audio server controls the audio device for a system. The module gets audio samples from the device and sends them to the appropriate module, and sends sound samples coming from other modules into the audio device. It also detects incoming calls and telephone hang-ups, and notifies the communications manager module on these events. The Basurde project has already produced audio servers for ISDN modems and for Dialogic analog telephone cards, both under Linux. During the development of Basurde[lite], the author produced an audio server for a sound card under Linux, for testing purposes, and substantially modified the ISDN server.

   The audio server, speech recognizer, and speech synthesizer modules work at a sample rate of $f_s = 8000$ Hz. The samples going to the speech recognizer are 16-bit linear, derived from 8-bit µ-law samples.

3. A speech recognizer converts a user utterance from sound samples to text written (more or less) in Spanish. It gets the sound samples directly from the audio server, not the communications manager.

   Depending on the capabilities of the speech recognizer, information about how sure the recognizer is about each recognized word (recognition confidence) could also be given. The actual module that we used cannot, but the rest of the system is prepared to handle these confidence values.

4. A natural language understanding module (NLU) converts a piece of text into *frames*, an internal representation of the meaning of that text (see 2.2.2 on page 19).

5. A dialogue manager (DM) responds to user input following a certain dialogue structure, thus guiding the conversation. Both its input and output are in frame format. It also generates a greeting as the first system utterance when an incoming call is accepted.

6. An oral response generator (ORG) converts the dialogue manager’s reply from frame format into Spanish text (with some annotations about how the text is to be read aloud).
7. A speech synthesizer (or text-to-speech program) converts the system's reply from text format into sound samples, that is, it "reads aloud" the system's reply. The sound samples are given directly to the audio server, not the communications server.

8. A module that acts as a local front-end to the train timetable database. This database is not directly available to the Basurde or Basurde[lite] development teams. However, it can be accessed through the railway company's web site. This module forms a web connection to get the desired information.

Of all these modules, the natural language understanding module, the dialogue manager, and the oral response generator are designed as part of the Basurde[lite] project. We have used the communications manager, speech recognizer, database front-end and speech synthesizer developed in the Basurde project, but we had to make some minor changes to them. Even if the audio server is not part of the Basurde[lite] project, the author developed an audio server for a sound card under Linux and substantially modified the Linux audio server for ISDN modems, so that the Basurde[lite] system could be tested.

Some of these modules require configuration files. The configuration files for the natural language understanding module, the dialogue manager and the oral response generator, which fully describe the translation process that these modules do, are designed as part of the Basurde[lite] project.

The speech recognizer requires information about what it is supposed to recognize. This is called the language model. This model describes the input language from the sentence level, through the word level, and down to the phonetic level. The models for the sentence and word level are part of the Basurde[lite] project. We used models for the phonetic level available from the TSC department at the Universitat Politècnica de Catalunya.

The models for the speech synthesizer configuration files were already available from the university.

### 2.1.2 Mixed C++/Tcl programming

Since describing the dialogue applications in configuration files is not a simple task, we saw that we would need to design a simple language for these configuration files, a trivial key=value structure for each line would not be enough. Given that we needed a non-trivial language, why bother designing one when we can use a full-fledged programming language as the language for configuration files?

One reason might be that this programming language does not support, or does not support in a convenient way, the operations that would be considered primitive for Basurde[lite]. The solution, of course, is to use this language
as a base for a domain specific language or DSL, and implement the missing primitives in, for example, C++. See [45] for a discussion on using existing languages for implementing DSLs.

Some parts of each module are written in the C++ programming language (using the Standard Template Library [46, 34]), and some are written in the Tcl programming language [36]. We used Norman Ramsey's novweb [42, 41] to combine documentation, and source code for the interface and implementation part in single files.

Tcl is an imperative scripting language based on text rewriting. It is not very powerful and does not scale well, but it is a good choice for rapid development and for adding programmability from the outside to static programs. Many languages are available for this function: Python and Perl to name a few. The simplicity of Tcl and the author’s prior experience with it was an important factor in choosing one.

Other architectures were also evaluated. The most promising ones were a Haskell-only implementation, or a DSL implementation using Scheme as the DSL host and C++ to implement the primitive operations. Given the AI-like nature of this project, these two options look nicer and more practical than the C++/Tcl option.

There are several reasons why they were not chosen:

Even if using a higher-level language such as Haskell or Scheme as the configuration language would mean simpler configuration files, it is not clear that they would be any easier to understand to a programmer not trained in functional languages. By using a simple imperative language, the typical dialogue application developer can more easily understand them.

There were concerns from the Basurde group about the speed of programs in these languages. In the end we saw that the only critical point was the NLU module, and that even the C++/Tcl combination (Tcl is not a very fast language) was fast enough.

There were also concerns about lack of portability and support. However, both Haskell and Scheme have ports, at least, to many flavors of Unix and to Microsoft operating systems. It is true though that the general lack of development tools recommends against a full-Haskell approach.

This means that a basic knowledge of Tcl is necessary to understand and modify the dialogue application. Fortunately, Tcl is simple and does not use any new ideas compared to, for example, C or Java. To allow people not versed in Tcl to understand some of the details of this document, a short Tcl tutorial is given in section B.8 (page 203).

We used the C API to the Tcl interpreter defined for version 7.6 or Tcl. In version 8.0 a new API was introduced, but the old one is still supported. We
chose to use the old API because it is much simpler. It is also slower, but this
was not a problem.

2.1.3 Structure of one Basurde[lite] module

The modules that have been developed as part of the Basurde[lite] project all
follow a common program structure. This is described graphically in figure 2.3
(page 17). Again, the parts developed in the Basurde[lite] project are shown
in light blue.

The central part of each module is an interpreter for the Tcl language. Among
other features, it glues together the remaining parts of a module.

![Diagram of Basurde[lite] module structure]

Figure 2.3: Structure of a generic Basurde[lite] module

Three program sections handle the communications with the rest of the Bas-
urde[lite] system. The Basurde libcom library handles the low-level com-
munications. Some programming glue provides Tcl interfaces to the libcom
library, so that its primitives may be called from a Tcl program running in
the Tcl interpreter. Finally, a piece of Tcl program uses these primitives to
to control the communication loops. This part is very similar (but not equal) in
all Basurde[lite] modules.

There is also a C++ block that provides the operations related to the dialogue
system that are not handled at the Tcl level, either because it would be too
slow, or because it would be too cumbersome\(^2\). A piece of C++ code provides
Tcl interfaces to these primitive operations. This part provides the specific
functionality of each Basurde[lite] module.

\(^2\)Each language is best applied to a certain kind of problem. We could of course program
the whole thing in Tcl or C++.
CHAPTER 2. STRUCTURE OF THE BASURDE[LITE] SYSTEM

The **main** C++ function is also provided here. It processes the command-line arguments, loads the Tcl interpreter and the several Tcl program files, and starts the communications loop.

In some modules, some “primitive” operations are actually defined in Tcl. These are automatically loaded from a text file when the program starts.

Finally, since the configuration files are actually programs they should be viewed as part of a module. They may provide some call-back functions (Tcl functions which will be called from the C++ side). Some modules actually require that some Tcl functions or variables be defined in these files.

The dialogue manager also calls the **exemple-renfe** program, which remotely accesses the timetable database.

### 2.2 Module interfaces

In this section we describe the interfaces between Basurde[lite] modules and between the modules and the operator. We describe the format of information at the various interface points and list the differences between the Basurde[lite] and the Basurde communications protocol. We finally describe the execution modes of the Basurde[lite] modules.

#### 2.2.1 Protocol stack structure

We use many stacked communication protocols, as shown in figure 2.4 (page 18).

![Diagram](image.png)

**Figure 2.4:** Communication interfaces and layers

The Basurde communication protocol, provided by the **libcom** library, runs on top of TCP/IP and provides some simple communication primitives that try to hide the details of the distributed nature of a Basurde system. We access this
2.2. MODULE INTERFACES

level through the API implemented in the libcom library, defined in [30]. This provides functions to build packets of a variety of kinds: packets containing recognized words, sent from the speech recognizer to the NLU module, packets containing raw audio data to and from the audio module, ...

The NLU, dialogue manager, and ORG modules access the libcom level through a Tcl command defined at the C++ level, the comm command. The usage guide for this command can be found in sections B.4.6 (page 179), B.5.11 (page 194) and B.6.4 (page 198). This level provides simple primitives to send arbitrary strings to other modules. For each module, it also provides means to send some specific control packets to other modules. For example, in the dialogue manager module there is a sub-command that instructs the whole system to terminate a call.

The programs defined in the application definition files use this Tcl command to communicate between modules.

2.2.2 Frames

The natural language understanding module must send to the dialogue manager a representation of the meaning of user utterances. Similarly, the dialogue manager must send to the oral response generator a representation of the meaning of the system's output.

For the Basurde project, it was chosen that frames were used as representations of meaning. Defined in [31], each frame is a unit of meaning. It can correspond to one or more sentences, and one sentence can correspond to one or more frames. The same structure is used for input frames, those going from the NLU module to the dialogue manager, as for output frames, those going from the dialogue manager to the ORG module.

These are not exactly the same as the "frames" used for knowledge representation in some Artificial Intelligence systems, because in the latter there are complex inter-frame dependencies and relationships, as well as internal and external constraints. Basurde frames are mostly independent units, and inter-frame references, if any, are implicit.

Each frame has a frame type which describes the intention of a user (for input frames) or a system (for output frames) utterance. Examples of intentions are: give information to the other party, ask a question, make an offer, or signal an understanding problem.

Each frame also has zero or more case-value pairs. Each such pair carries a piece of information related to this meaning unit. The case (sometimes called case name) acts as a label for the value (sometimes called case value), that is, the case tells what kind of information the value is about. Examples of cases are: arrival city, price of one-way ticket, departure time, departure date.
While the Basurde project defines in [44, 31] a set of frame types and cases, during the detailed development of Basurde[lite] we found out that some were inadequate, and that we needed to add some more frame types and cases, and give new meanings to some of them. The main reason for this is that in Basurde the user has the dialogue initiative, whereas in Basurde[lite] it is the machine who has it. Therefore different kinds of sentences are likely to be exchanged. This is detailed in chapter 5, where we give the full list of valid input and output frames.

**Basurde interchange format**

Frames are transmitted using the Basurde protocol as packets of type `FRAME_COMPREHENSION` (from the NLU to the dialogue manager) and `FRAME_ORAL` (from the dialogue manager to the ORG). A list of zero or more frames is encoded in one such packet as a formatted string, in the following way:

1. The encoding for a list of frames is an open bracket “{”, followed by the encodings for each frame in the list, followed by a close bracket “}”.

2. The encoding for a frame is an open bracket, followed by the frame type as a string, followed by the encodings for each case-value pair in the frame, followed by a close bracket.

   The string encoding for a frame type is always a single word containing no brackets.

3. The encoding for a case-value pair is an open bracket, followed by the case as a string, followed by a space, followed by the value as a string, followed by a close bracket.

   The string encoding for a case is always a single word containing no brackets. The string encoding for a value is not so clearly defined by Basurde. In Basurde[lite] we accept anything that does not contain a close bracket.

4. Spaces can be added before open brackets and after close brackets, for extra visual comfort.

Frames encoded in that way have a striking resemblance to LISP trees.

This is the format that should be used when calling Basurde communication functions, and the Tel `comm` command. This is a Basurde standard; throughout this document we call it *Basurde interchange format*. 
Tcl frame format

Frames in the Basurde interchange format cannot be easily manipulated by Tcl programs. For this reason, there is an alternative representation of frames, more suited to Tcl. We call it *Tcl frame format*.

Although we use the Basurde interchange format to interface with the `comm` command, we provide Tcl functions that translate between the two formats.

1. A *list of frames* is a valid Tcl list. Each element of this list is a *frame* in Tcl format.

2. A *frame* is a valid Tcl list. The first element of this list is the frame type. The remaining elements, if any, are to be taken two at a time (even if this grouping is not explicit in the list structure). Each pair represents a case-value pair. The first word in the pair is the case name and the second word in the pair is the case value, or attribute (as in `name1 value1 name2 value2 ...`).

\[
\text{list-of-frames} \rightarrow \{ \text{frame}^* \} \\
\text{frame} \rightarrow \{ \text{frame-type case}^* \} \\
\text{case} \rightarrow \text{case-name case-value}
\]

As usual in Tcl lists, quoting rules allows building lists whose elements contain spaces.

Textual representation

In this document, we represent frames in rectangles. The frame type is given in **boldface** as the first line of text in the rectangle. Each subsequent line is a case-value pair (sometimes a long case-value pair wraps to the following line). The case is given in **typewriter** type. The value may be given in normal type (for literal values) or in *italic* type (for variable values).

Some frames have a ( ... ), which means that additional case-value pairs may be found in that frame. We use this feature in the ORG module in an attempt to make the Basurde[lite] ORG more or less usable as the Basurde ORG. With perhaps some minor changes in the application definition files, the Basurde[lite] ORG can accept frames from the Basurde dialogue manager and generate text for them; probably not the whole meaning of these frames will be captured, however.

For example, this is the Basurde, Tcl and textual representations of a frame\(^3\) that means that the user wants to tell the system that the departure place is

\[^3\text{For the Basurde representation, we give that of a list of frames containing only one frame.}\]
Barcelona, and that the arrival place is Tarragona, using hypothetical values for the frame type, case name and case value:

\[
\text{Basurde (user\_information} \\
\quad \text{(dep\_place Barcelona)} \\
\quad \text{(arr\_place Tarragona))}
\]

Tcl \{user\_information dep\_place Barcelona arr\_place Tarragona\}

<table>
<thead>
<tr>
<th>text</th>
<th>user_information</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dep_place: Barcelona</td>
</tr>
<tr>
<td></td>
<td>arr_place: Tarragona</td>
</tr>
</tbody>
</table>

### 2.2.3 Changes to the Basurde communications protocol

We also found that we needed some packet types and functionality that the Basurde communications protocol did not provide. We made the following modifications to the protocol:

1. In the Basurde protocol, when the audio server detects an incoming call, it sends a packet of type LAMADA\_ENTRANTE to the communications manager. The communications manager will then generate an initial greeting and send it, in text form, to the speech synthesizer.

   This was not adequate for our module structure. For Basurde[lite], the audio server also sends this packet to the communications manager. However, the latter does not generate the initial greeting. Instead, it sends LAMADA\_ENTRANTE packets to the NLU, dialogue manager, and ORG modules. Upon reception of this packet, the dialogue manager generates an initial greeting and sends it, as a frame, to the ORG module (which will convert it to text form and send it to the speech synthesizer).

2. Something similar happens with call end detection. In the Basurde protocol, when a call ends, the audio server sends a packet of type FIN\_LLAMADA to the communications manager.

   We modified the communications manager so that this packet is retransmitted to the NLU, dialogue manager, and ORG modules.

   These modules will reset themselves and prepare for a new call when they receive one such packet. Also, they will ignore all data packets until they receive a LLAMADA\_ENTRANTE packet.

3. We needed a way to send control information from the dialogue manager to the NLU, speech recognition, and audio modules. For example, the dialogue manager uses its knowledge about the current dialogue situation to tell the speech recognizer what kind of sentences it should expect.
The Basurde protocol does not have a packet type for this kind of data. We chose to reuse the FRASE type (which should normally be used for sending utterances in text form to the speech synthesizer) for this new purpose because:

(a) That packet type was not already in use for transmissions from the dialogue manager to these modules and

(b) A FRASE packet normally carries data as a human-readable string of characters, and the control data that we will sent through them is also in human-readable string form.

We use the following control strings as data for these packets:

(a) set_state grammar (sent to the speech recognition module). Sets the root recognizer grammar to be grammar. This must be the name of a top-level grammar defined in the recognizer grammar file. This prepares the recognizer to recognize a specific kind of expressions (for example, only dates, or only answers to yes/no questions).

(b) timeout time (sent to the speech recognition module). Sets the time, in milliseconds, that the recognizer will wait before sending a timeout notification if the user does not say anything.

(c) set_state staterescr (sent to the NLU module). Tells the NLU module which kind of expressions it should expect, and gives it additional information such as the reference date (what should be used for expressions like “one day later”).

(d) do not record (sent to the audio server). Instructs the audio server not to change to record mode (where user speech is captured and sent to the speech recognizer) after the next system utterance is sent to the user. This is needed because we sometimes need the system utterance to be produced by more than one frame list.

Refer to section B.5 (page 181) for more details.

Basurde specifies two operating modes for audio servers: raw samples and parameterized. The audio server that was developed during this project can only run in raw samples mode.

2.2.4 Module invocation interface

All three Basurde[lite] modules (NLU, ORG and dialogue) consist of a large executable, several Tcl files that describe the dialogue application, and a “Tcl library” file (usually named ...lib.tcl). Some modules may also have additional files, such as data files or dynamically linked libraries.
The name of a main Tcl application definition file must be given to the program on the command line. This main Tcl file should include any other necessary Tcl application definition files (for example, using the Tcl command source). If the name of this file is not given, the program will issue a warning but start nevertheless, with an “empty” application.

The name of the Tcl library file, however, need not be given on the command line. There is a default name for each module. An alternate name can be given on the command line, or the program can be instructed not to load any Tcl library file. This file is supposed to be largely application-independent. It contains small helper functions, some basic operations which are defined in Tcl instead of in C++, and, most importantly, the procedures that implement the communications loop (which is the part of the program that listens for incoming packets and gives the appropriate response). Some of the procedures defined in this file are called from the C++ side, so the module may not work if this file is replaced by another one with some missing definitions.

These modules can run in 3 different modes: stand-alone, semi-stand-alone, and manual. The running mode can be selected through the command line.

The stand-alone mode is the normal mode for production systems.

The other two modes are used for debugging tasks. In the semi-stand-alone mode, there is a modified communications loop. Some control packets will be automatically processed, but for some other packets a message will be given to the user and the program will enter Tcl console mode, in which an operator can issue arbitrary Tcl commands. The operator should explicitly respond to these packets. This mode is useful for debugging problems related to inter-module communication.

In manual mode, there is no communications loop. The program is always in Tcl console mode, and the operator must, if she wants to, receive and send packets herself. This mode is useful for debugging problems related to the translation process, not for communication problems.

For an explanation about command-line arguments and other details about running the Basurde[lite] modules, see appendix A (page 153).
Chapter 3

Designing the dialogue application

Designing the dialogue application involves deciding how typical dialogues should look like, formalizing these dialogue patterns in a suitable form, deciding which expressions the system should understand and what the system should say, find a mapping between these expressions and Basurde frames, and finally producing the application definition files. The design process is described in more detail in 3.1.6 (page 38).

This chapter describes the dialogue strategy, and the input and output expressions supported by Basurde[lite].

3.1 Overall dialogue strategy

3.1.1 Dialogue goals

This system provides information about long-distance trains of the RENFE Spanish railroad system. Following the Basurde goal definition, it is not possible to make reservations or ask about the intermediate stations that a train calls at.

The user must specify the departure and arrival place (as a city or station name) of the trip, and the departure or arrival date (or, for some special cases, a date range). Only trips needing only one train (without connections) can be handled with the system. These restrictions arise from similar restrictions in the RENFE timetable database that the system uses. Given that queries to the database are rather slow (5 to 15 seconds per query) and that in each query we get only all the direct trains on a certain date for a given departure and arrival place, we decided not to handle routes requiring the passenger to change trains.

25
CHAPTER 3. DESIGNING THE DIALOGUE APPLICATION

More precisely, the user must provide the system with:

1. The departure city or the departure station, and
2. the arrival city or the arrival station, and
3. the departure date, or departure date range, or departure day type (work-day or holiday), or arrival date, or arrival date range, or arrival day type.

Given this information, the system can give the following information for all trains on that route and date:

1. Departure time,
2. arrival time,
3. train type,
4. services offered on the train (restaurant, telephone, ...), and
5. ticket price, for all ticket classes, and both for one-way and return tickets.

The Basurde system is supposed to be able to give the departure station and arrival station if the user gave the departure or arrival city. However, the RENFE database does not give us this information.

In a visual-based information access system, we could just provide all that information in a single list and let the user find what she is looking for. However, in a voice-based system we must find ways to progressively reduce the data resulting from a database query, so that the user is able to handle it in voice form.

We decided that information output should be limited, by ergonomic reasons, to the following:

1. To provide the user with additional feedback, and to refresh her memory, we will give the departure and arrival place, and the departure date, whenever we give information about one or more trains.

2. We consider that lists of more than three trains (long lists) are too large for the user to keep in memory. Therefore, we will not give information for all trains in these lists. We will give only the length of the list, and the departure and arrival times of the first and last train in the list. Also, we will try to reduce these lists using criteria that do not relate to the position of trains in the list.
3.1. **OVERALL DIALOGUE STRATEGY**

3. For lists of two and three trains (*short lists*), we give the departure and arrival times, and the train type, for each train. We will try to reduce these lists using criteria that relate to the position of trains in the list.

4. If we manage to reduce the result to only one train, we will give the departure and arrival time for the train, unless the user asks for more information. If the arrival date is different from the departure date, both will be given.

5. The full information for a train will only be given for a single train per dialogue turn, and only if the user explicitly requests it.

When the database query returns more than one train, we let the user choose among several *reduction criteria*, reasons why she would choose one train or another. The reduction criteria that we accept for long lists are the following:

1. Departure time (e.g. quarter past two) or time range (e.g. before three o'clock),

2. arrival time or time range,

3. train type,

4. one service offered on a train (no more than one),

5. select the fastest train (the train with smallest trip time), and

6. select the cheapest train.

For short lists, we accept the following:

1. A position in the list (e.g. the first train, the second one, ...), and

2. select the train that leaves or arrives earliest,

For both kinds of lists, after we have managed to get to a single train, we are offered to move to the next or previous train (what *next* and *previous* means exactly depends on the context, and will be fully described in chapter 5). This is analogous to list scrolling in a GUI environment.

Also, when the user gives a date interval instead of a single date as departure or arrival date, we tell her how many trains we have for each date in the interval and let her choose a specific date.
3.1.2 System ergonomics and usability

It is very important to follow some usability guidelines when designing human-machine dialogue systems (as in any other human-machine interface). The limited short-term memory and attention span of human beings, the impossibility to present the whole information at the same time, and the fact that speech interfaces are often used by inexperienced people, make these systems particularly prone to confuse its users.

We tried to follow the design guidelines given in [8, 48, 43], with some modifications since ours is a simple machine-driven system limited to one task, not a multiple-task mixed-initiative system. In particular:

1. We try to keep the user informed of the current dialogue situation, of exactly what she is allowed to do from that point, of exactly what kind of answers the system can accept at each dialogue turn, and of what the system thinks that the user wants.

2. We inform the user of the system’s limitations.

3. We give the user feedback of what the system has understood (using implicit confirmation).

4. We provide the user with navigation tools to examine pieces of information which are too large to give in a single block.

5. We provide the user with tools to change the current task, and feedback on what the current task is. However, only a very limited range of high-level tasks is available. At a lower level, the user can answer some questions in a way that effectively triggers a task change (such as giving the arrival station when the system asks for the departure city).

6. We decided that the system would always have the dialogue initiative. This is not a user-driven or mixed-initiative system. We designed the dialogue strategy and the specific questions so that the user is not tempted to try to gain the initiative.

7. We provide means for error detection and correction: the system can take appropriate action when it believes that there was a recognition error, and the user is given tools to correct errors made by the system, or to change her mind. However we provided little mechanisms for the system to detect user inconsistencies.

8. We try to detect when the dialogue is going astray and fall back to less comfortable but more robust dialogue sequences. While we do not use hierarchic prompts as suggested by [47], we turn implicit confirmations into explicit confirmations.
3.1. **OVERALL DIALOGUE STRATEGY**

We also provide a last-resort mechanism that the user can activate to bring the dialogue to a known, “safe” state in case the system (or the user) gets completely confused.

Finally, if the system detects too many errors, it gives up and tries to transfer the call to a human operator. Even if in these cases the dialogue system will have failed, at least we will increase the final user satisfaction.

9. We play a short beep after each system dialogue turn to indicate the user that the system is ready to listen. In that way we try to avoid user barge-in, user interruptions while the system is talking.

The system does not allow user barge-ins. Low system performance may occur if still the user interrupts the system, so it is better to discourage the user from doing so.

However, we did not include any help mechanism (even if recommended by [48]) other than optionally giving the operating instructions at the beginning and providing a mechanism for user-initiated correction and recovery.

### 3.1.3 Confirmation strategy

Since the recognition success rate is, of course, lower than 100%, we need to find a way to check the correctness of the recognized information which, at the same time, does not make interaction much slower.

To check the correctness of a piece of information the dialogue system will use two kinds of confirmation requests: implicit and explicit. In explicit requests we make an explicit question to the user, asking her to say if we understood her correctly (“*Do you really wish to leave tomorrow?*”). Implicit requests are piggy-backed on other kinds of system output; they are designed so that the user does not have to do anything if the recognition was correct, but may tell the system if the recognition was wrong (“*You wish to leave tomorrow. At what time?*” is an implicit confirmation request for the departure date, and a normal question for the departure time). Following [28], we do not try to join an implicit confirmation request and the remaining information into a full sentence, as this tends to distract and confuse the user (for example, “*At what time do you wish to leave tomorrow?*”).

We have two methods of evaluating how well the dialogue is currently going: Recognition confidence values from the speech recognizer tell us about the quality of the current user reply. The records that we keep about past user-initiated corrections help us determine the overall dialogue quality. Recognition errors increase the current error level. User-initiated corrections also increase it, with different amounts. The error level decreases over time, so that distant errors are ignored. Whenever the error level is above a certain threshold, all confirmation
requests are explicit even if they would normally be implicit. Above a higher threshold, the system gives up and transfers the call to a human operator.

We have two confidence thresholds which are generated by the NLU module after word-wise confidences given by the speech recognizer. Frames below the low threshold are marked as “unrecognized” and discarded: the system will ask the user again. Frames above the high threshold are accepted (but not always confirmed): the system may ask the user for confirmation using and implicit confirmation request. Frames between the two levels are accepted but the system will use explicit confirmation requests for them.

In order to reduce the recognized language (and thus improve the recognition success rate) we do not allow the user to correct the value asked to confirm in an implicit confirmation. Only the negation is detected, and the system makes a full question on the next turn. For example, if the system says “From Barcelona. What date?” and the user wants to correct the departure city (Barcelona) to, say, Tarragona, by saying “No, Tarragona”, only the “no” will be understood. The system will then ask “What is the departure place?”.

Since implicit confirmation requests may be answered with a “no”, we cannot join one such request with a “yes/no” question. We provide the application developer with mechanisms to force the resolution of all pending confirmations before asking a “yes/no” question.

It would be interesting to detect and use the user’s positive and negative cues as described in [27] but that would make the system too complex for this project. However, the dialogue manager is modular enough that this feature can be added afterwards.

3.1.4 Technical constraints

The system, like its parent Basurde system, does not support user barge-in, user interruptions while the system is speaking. This is a big limitation because the natural way for the user to correct a system misunderstanding is to interrupt it quickly after the user realizes she has been misunderstood. Because of that limitation, we had to use an alternate confirmation scheme including explicit and implicit confirmation questions.

We designed the Basurde[lite] modules assuming that the speech recognizer would be able to give recognition confidence values for the recognized words or sentences. We use these values to decide when to use explicit or implicit confirmation, or when to ask the user to repeat her utterance. We also keep a record of part recognition errors to bias the confirmation questions to more explicit ones. However, the speech recognizer is actually not able to give recognition confidence values for the recognized words or sentences. We decided to keep using past errors to decide whether we should use explicit or implicit confirmation, to use implicit confirmation as the default confirmation method
3.1. OVERALL DIALOGUE STRATEGY

for normal questions, and not to use any kind of confirmation for very simple questions (such as yes/no questions) where we can expect the recognition success rate to be very high.

Also, there is no explicit provision in the recognizer for detecting out-of-vocabulary words. We included an “out-of-vocabulary” sub-model in the language model that would catch these words and signal them to the NLU module.

The system was, at least theoretically, capable of using system barge-in, that is, interrupting the user while she is talking. This is useful to provide positive feedback to the user (expressions such as “yes”, “I see”, “uh-huh”. We might do that because the speech recognition module issues the string of recognized words not all at a time, at the end of the recognition process, but in smaller chunks of maybe two or three words.

However, there were some problems: The recognizer is too slow to effectively interrupt the user. Also, the Basurde architecture and its audio servers currently handle audio I/O in a half-duplex way, that is, the system either listens to the user or talks to the user, so we would miss the part of user speech that overlaps the system barge-in. Because of these technical problems, and because system barge-in is ergonomically risky and difficult [21], we decided not to use it.

In the end we developed an NLU module that needs the whole user utterance in order to begin processing it. Since system barge-in had already been rejected, we did not try hard to make an NLU that could process user input in smaller chunks.

The speech synthesizer produces voice which is clearly computer-made. We believe this to be an advantage rather than a drawback. People know that computers are “not very intelligent”, and most are used to DTMF telephone dialogues, or other equally simple systems. We hope that this will induce the users to use only simple and short sentences, and to lower their expectations as to what the system can handle.

3.1.5 Dialogue graphs

In this section we describe in graphical form the conversations that can be held with the Basurde[lite] system. We use a directed graph which can be mapped to a finite state automaton, to represent all possible conversations.

Nodes represent system utterances. There are two kind of nodes: those shown as a hollow circle represent system questions to the user; those shown as a circle with a square in it represent system output which is not a question. A description of the utterance is given next to the node.

Graph transitions (arcs) represent user responses. A description of the response is given in sans serif type next to the arc. Of all arcs going out of a node, the
dialogue automaton takes the one that corresponds to the user’s reply. An additional condition to following an arc may be given in normal type next to the arc.

We also have null nodes (shown as smaller hollow circles) and null transitions (without a user response). These are included to make the drawings clearer but do not affect the dialogue.

The initial state is marked by a $\forall$ symbol pointing to it. The final state is a node drawn as a smaller circle with a cross in it ($\otimes$). The whole graph would not fit on a single page: by using expressions like “go to N0” and rounded boxes with labels like “N0” inside we link parts of the graph which are spread over several pages.

Some nodes have “state names”, small labels near them, or a short name inside the circle. These names are used in the text description.

The dialogue graphs do not fully describe the sequence for a conversation, because at a given time the system is not only asking a question, but also waiting for confirmation of a previous answer. This effectively gives us two current locations. Transforming this pair of graphs into a single graph (for example, having as states the Cartesian product of the original graphs’ states) would give an unwieldy graph. Therefore the confirmation strategy, orthogonal to the main dialogue strategy, is not shown here. However, a double thick line crossing an arc ($\parallel$) indicates that the dialogue cannot go through this transition if there are pending confirmations.

Also, questions about dates and places are more complex than what is shown here. The user can usually explicitly tell whether it is a departure or an arrival date (or place), and can change at will between the different kinds of dates (single date, date interval, or day type) and times (single time, time interval). For times, there is also the complication that AM/PM-ness (whether a time is AM or PM) may be separate from the numerical time.

In figure 3.1 we show a general overview of the whole dialogue graph. This is mainly an aid to find the way in the detailed graph drawings.

Figure 3.2 contains the top level dialogue automaton, the initial and final states, and the questions that are asked before the database is queried.

Figure 3.3 describes how the system handles short lists, those with two or three trains.

Figure 3.4 describes how the system handles long lists and other situations where we offer the user the same choice of reduction criteria.

In figure 3.5 we describe how to handle a single train. This includes giving the information we have for the train, and also offering to select another train from the previous result (using list scrolling and selection by position).

Finally, figure 3.6 describes what we do when the current query has finished,
### 3.1. OVERALL DIALOGUE STRATEGY

![Diagram overview of dialogue strategy](image-url)

- **Main**
- **Query database**
- **Process one train**
- **Choose train in short list**
- **Reduce query using train attributes**

**N0**: Want info on other trains?

**N1**: Process one train

**N23**: Choose train in short list

**CRIT**: Reduce query using train attributes

**Goodbye**

**Figure 3.1: General dialogue overview**
Figure 3.2: Top-level dialogue automaton
Figure 3.3: Handling short lists
Figure 3.4: Handling long lists
3.1. OVERALL DIALOGUE STRATEGY

Figure 3.5: Handling a single train
either because there were no trains for that date, or because the user does not want more information about them. We ask the user if she wants the system to make another database query.

![Diagram](image)

**Figure 3.6: Offer to look for other trains**

You may wish to look at the two sample dialogues transcribed in section 6.4 (page 142).

### 3.1.6 Design method

We did not design the whole dialogue application in one big step. Instead, we followed this approach:

1. We designed an initial version of the dialogue graphs, based on intuition and informal “simulations” where coworkers were asked to conduct conversations with a person who behaved as the dialogue graphs dictated (the conversation was fully human-to-human; this is not a Wizard of Oz experiment).

2. Then we listed all the expressions that the typical user would use in answering the system’s questions, and which the system would have to
understand. We also decided precise wordings for each system utterance. The following step was to map these user and system utterances to Basurde frames, and propose new frames for those expressions that could not be mapped into standard Basurde frames.

3. As we designed the rest of the dialogue application (input and output sentences, input and output frames, and eventually the application definition files) we refined the dialogue graphs.

4. As soon as the three Basurde[lite] modules (NLU, dialogue manager, and ORG) became available, we tested the application with a text-only interface: User input was given to the NLU in written form, and the output from the ORG was shown on a computer terminal, not sent to the speech synthesizer. We used this step to debug the programs and the dialogue application. Some ergonomy problems were detected and solved at this point.

5. When we could finally start the whole Basurde[lite] chain we tested the whole system. Apart from debugging several programming problems, we decided that, in order to get better navigation capabilities and make the interface more intuitive, we needed to do a major overhaul of the upper levels of the dialogue strategy, while keeping the lower levels (date, time, place and confirmation handling) as they were.

6. After we implemented and tested those changes, we corrected some remaining minor ergonomy problems.

7. Finally, we proceeded to the evaluation phase, described in chapter 6 (page 125).

3.2 Output sentences

We analyzed the dialogue graphs to determine which utterances the system should be able to generate. This section lists them. They are given in Spanish, with some variable parts in emphasized type. An English translation and an explanation of some of the frames is also given.

We took care, in choosing the system utterance, of the effect that they have on the user's response. Studies [26] show that users adapt the length and complexity of their responses to that of the system's questions. Others show that users tend to adapt to the words and structures used by the system. See [19, 6] for a summary of these results. Therefore we chose expressions which were simple and short, even to the point of making them grammatically incomplete or unacceptable in written Spanish. We did not include long and convoluted politeness expressions that are
40  CHAPTER 3. DESIGNING THE DIALOGUE APPLICATION

sometimes found in these systems. Yes/no questions (those that are normally answered with “yes” or “no”) are constructed in such a way that it is obvious to the user that the system expects a yes/no answer. For multiple-choice questions, we usually give the full list of valid options in the form we would like to user to use (even if we may accept other forms).

From this list and the Basurde standards about frames [31, 44] we decided how we could map these utterances to existing Basurde output frames, and which new frames had to be added to the standard.

Note that many utterances are given in three forms: full question, explicit confirmation, and assertion. This corresponds to the three dialogue modes of the Basurde[lite] system.

- Horarios RENFE, buenos días. Welcome to RENFE timetable information.
- Adiós, y buen viaje. Goodbye, and enjoy your trip.

- ¿Desde dónde desea viajar?
- ¿Desea salir de place? Where would you like to leave from?
- De place. Would you like to leave from place?
- ¿Cuál es su destino? Leave from place.
- ¿Desea ir a place? Where would you like to go to place?
- A place. Go to place.

- ¿Qué día desea salir?
- ¿Desea salir date? On what date would you like to leave?
- Salir date. Would you like to leave on date?
- ¿Qué día desea llegar? Leave on date.
- ¿Desea llegar date? On what date would you like to arrive?
- Llegar date. Would you like to arrive on date?

- Un momento, por favor. One moment please.

- ¿Quiere que le explique cómo funciona el sistema?
- (give some brief hints about how to use the system) Would you like to hear the operating instructions?
3.2. OUTPUT SENTENCES

- ¿Desea información sobre otros trenes, sí o no?  
Would you like information about other trains?

- ¿Quiere información del primer (segundo, tercer, ...) tren?  
Would you like information about the first (second, third, ...) train?

- ¿Quiere información del anterior (siguiente) tren?  
Would you like information about the previous (next) train?

- ¿El ordinal?  
The ordinal? (Note that this can also be “next” or “previous”)

- El ordinal.  
The ordinal. (Note that this can also be “next” or “previous”)

- ¿Quiere más información de este tren, sí o no?  
Would you like more information about this train?

- ¿Le miro el tren siguiente o anterior?  
Should I check the previous or next train?

- ¿El siguiente, o el anterior?  
The previous or the next one?

- ¿Del mismo viaje, el viaje de vuelta, u otro viaje?  
About the same trip, the return trip, or another trip?

- Desea buscar un tren según: su horario, los servicios que ofrece, el tipo de tren; ¿lo quiere el más barato?.  
You want to select a train based on: its timetable, the services offered on it, the kind of train, or you want the cheapest train.

- ¿El más barato?  
The cheapest train?

- El más barato.  
The cheapest train.

- Hay trenes con service-list. ¿Qué servicio le interesa?  
There are trains with service-list. Which service are you interested in?

- ¿Un tren con service?  
A train with service?

- Un tren con service.  
A train with service.

- Hay trenes train-type-list. ¿Qué tipo le interesa?  
There are train-type-list trains. Which kind of train are you interested in?

- ¿Un train-type?  
A train-type?

- Un train-type.  
A train-type.

- ¿A qué hora desea salir?  
At what time would you like to leave?
¿Desea salir time?
Salir time.
¿A qué hora desea llegar?
¿Desea llegar time?
Llegar time.
¿El que sale antes?
El que sale antes.
¿El que llega antes?
El que llega antes.
¿Desea el tren más rápido?
El más rápido.

El tren va de d-place a a-place, sale d-date d-time y llega [a-date] a-time.
Sale de d-station y llega a a-station.

Es un tren train-type.
Cuenta con train-service-list.

El precio del billete no está disponible.
En ticket-class, el billete vale number pesetas [para sólo ida, y number para ida y vuelta].
El billete vale number pesetas [para sólo ida, y number para ida y vuelta].

El ordinal [es un train-type] sale d-time y llega a-time.

There are number [equally fast] [train-type] train[s] [with train-service] [leaving [d-date] [d-time]] and [arriving [a-date] [a-time]].

Would you like to leave at time?
Leave at time.
At what time would you like to arrive?
Would you like to arrive at time?
Arrive at time.
The earliest-leaving train?
The earliest-leaving train.
The earliest-arriving train?
The earliest-arriving train.
Would you like to take the fastest train?
The fastest train?
The fastest train.

That train travels from d-place to a-place, leaves d-date d-time and arrives [a-date] a-time.
It leaves from d-station and arrives at a-station.
It's a train-type train.
The services train-service-list are offered on it.
The fare price is not available at the moment.
In ticket-class, a ticket costs number pesetas [one way, and number for a return ticket].
A ticket costs number pesetas [one way, and number for a return ticket].

The ordinal train [is a train-type] leaves at d-time and arrives at a-time.
3.2. OUTPUT SENTENCES

- No hay ningun tren [train-type] [con train-service] [de d-place] [a a-place] [que salga [d-date] [d-time]] [y] [que llegue [a-date] [a-time]].

- Perdone, pero me he estropeado. Le paso con un operador.

- Disculpe, pero necesito más información para poder ayudarle. Por favor intente responder todas mis preguntas.

- Lo siento, pero no consigo entenderle. Le paso con un operador, no cuelgue.

- One random choice among “¿Oiga?”, “¿Hola?” and other similar expressions

- One random choice among “¿Cómo dice?”, “¿Puede repetir?” and “No le he entendido, ¿puede repetir?”

- Repito.

- ¿Salir time de am-pm or de am-pm? Leave at time AM or PM?

- ¿Llegar time de am-pm or de am-pm? Arrive at time AM or PM?

- ¿De la mañana o de la tarde? AM or PM?

- De am-pm-name. At am-pm-name.

- ¿Ha dicho am-pm-name? Did you say am-pm-name?

3.2.1 Date and time expressions

We give here a list of the expressions that the computer can produce for dates and times. This list should be taken as an informal reference; see the formal specification of date expression formats in section 5.4.4 (page 106) and 5.4.5 (page 108), and that for time expression formats in 5.4.6 (page 110) and 5.4.7 (page 111).

For expressions involving word classes (for example, month names), we give an example using only one word in the class.
The expressions for dates and date intervals are:

- el día tres  
- el tres de enero  
- el tres del uno  
- hoy  
- mañana  
- pasado mañana  
- el miércoles  
- el miércoles día tres  
- el miércoles tres de enero  
- el próximo día tres  
- el próximo tres de enero  
- el próximo tres del uno  
- el próximo miércoles  
- el próximo miércoles día tres  
- el próximo fin de semana  
- el fin de semana del día seis  
- el fin de semana del seis de enero  
- del ... al ...

- the 3rd  
- January 3rd  
- 01-03  
- today  
- tomorrow  
- the day after tomorrow  
- Wednesday  
- Wednesday 3rd  
- Wednesday, January 3rd  
- (next 3rd)  
- (next January 3rd)  
- (next 01-03)  
- next Wednesday  
- (next Wednesday 3rd)  
- next weekend  
- (the weekend of the 6th)  
- (the weekend of January 6th)  
- from ... to ...

The expressions for times and time intervals are:

- las seis menos cuarto  
- las seis menos cuarto de la tarde  
- las dieciséis cuarenta y cinco  
- las cinco cuarenta y cinco de la tarde  
- de las ... a las ...  
- antes de las ...  
- después de las ...  
- hacia las ...  
- sobre las ...

- quarter to six  
- quarter to six in the afternoon  
- seventeen forty-five  
- five forty-five in the afternoon  
- from ... to ...  
- before ...  
- after ...  
- around ...  
- around ...

### 3.3 Input sentences

We also analyzed the dialogue graphs to determine which kind of user utterances the system should be able to understand.

One of the advantages of having machine-driven dialogues is that, at each user turn, the machine knows what kind of reply to expect. For example, at a given point the machine may expect a time from the user. Or a place. But, if the dialogue is properly designed, there will probably not be a point in which the user can say both a place and a time. This allows us to focus the speech
3.3. INPUT SENTENCES

<table>
<thead>
<tr>
<th>dialogue situation</th>
<th>recognizer state</th>
</tr>
</thead>
<tbody>
<tr>
<td>a place *</td>
<td>_ciutat_estacio</td>
</tr>
<tr>
<td>a date *</td>
<td>_data</td>
</tr>
<tr>
<td>a time *</td>
<td>_hora</td>
</tr>
<tr>
<td>a part of the day (or AM/PM expression) *</td>
<td>_am_pm</td>
</tr>
<tr>
<td>an answer to a yes/no question</td>
<td>_si_no</td>
</tr>
<tr>
<td>a reduction criterion *</td>
<td>_crit_red</td>
</tr>
<tr>
<td>a train type *</td>
<td>_tipus_tren</td>
</tr>
<tr>
<td>a service offered on a train *</td>
<td>_serveis_tren</td>
</tr>
<tr>
<td>selecting a train in a list *</td>
<td>_sel_llista</td>
</tr>
<tr>
<td>a kind of trip (same, return, other)</td>
<td>_viatge</td>
</tr>
</tbody>
</table>

Table 3.1: Recognizer states

recognizer into a smaller set of possible input utterances. We expect that this will increase the recognizer success rate.

So, before deciding the exact utterances which should be accepted, we first determined which dialogue situations the user can be in. Each of these dialogue situations corresponds to a certain recognizer state and will have an associated recognition grammar.

3.3.1 Recognizer states

On table 3.1 (page 45) there is the list of recognizer states, together with the name of the top-level grammar that defines them.

For those states marked with *, there are also states having _cexp_ and _cimp_ as prefix (instead of __), which are used for explicit and implicit confirmation requests.

Also, there are some utterances which are recognized in all states. These are expressions which signal communication problems or convey meta-communication information or requests. For example, “Could you repeat?”.

3.3.2 Recognized sentences

For each recognizer state, we give a list of user expressions that should be recognized. We also give a list of expressions which should be recognized in any state. This list is complete as far as meaning is concerned, but the actual recognizer might accept slightly different expressions, or may also accept other equivalent expressions. Refer to the recognizer grammar file gram_recon/gram-bnf.txt (C.1.1 on page 207) for the actual grammars.
The following expressions are accepted in all states:

- ¿me lo puede repetir, por favor?  
  Could you repeat please?
- ¿... repetir?  
  Could you repeat?
- ¿cómo dice ... ?  
  Sorry?
- ¿cómo?  
  Sorry?
- ¿qué dice?  
  I don't understand you.
- no le entiendo ...  
  I couldn't understand you.
- no le he entendido ...  
  I'd like to talk to a person.
- desearía hablar con una persona  
  I'd like to talk to an operator.
- páseme con una operadora  
  Correction. (request to restart the dialogue)

The expression “¿qué?” or “Sorry?” was removed because it was too short to be confidently recognized.

In some states, apart from expressions carrying the information expected for that state, some expressions meaning “I don’t know” or “I don’t care” are also accepted. These are marked by (don’t know) in those lists. They are:

- no sabría qué decir  
  I don't know what to choose.
- no sé ...  
  I don't know.
- no lo sé  
  I don't know.
- me da igual  
  I don’t care.
- cualquiera ...  
  Any or them.
- ni idea ...  
  I have no idea.

Also, in most states we allow some rather meaningless expressions to prefix the actual reply. These are filler expressions that indicate that the user is thinking or doubting. Some of them, appearing on their own, do have a meaning, so they are usually not accepted alone:

- no sé ...  
  Dunno ...
- no lo sé ...  
  I don't know ...
- que  
  ( ... )
- quiero ...  
  I want
- le digo que ...  
  I said
- pues ...  
  Well
- ...  
  ...

And also in most states we allow the user to give a politeness expression after the actual reply:

- gracias  
  Thanks
3.3. **INPUT SENTENCES**

> por favor

*Please*

In some states we do not accept any of these prefixes. In some other states, the accepted prefix set may be slightly modified to better match the response expected in the state. Refer to the grammar file for more information.

**Expressions accepted when expecting a place:**

- **place-name**
- A **place-name**.
- De **place-name**.
- Quiero salir de **place-name**.

The direction clues given by “a” and “de” is ignored.

**Expressions accepted when expecting a date:**

- [marchar I irme I salir] **date-expr**
- Llegar **date-expr**
- [marchar I irme I salir] **date-expr**
- Llegar **date-expr**
- [date-expr ... [marchar I irme I salir]]
- [date-expr ... llegdar]

However we found out the short departure/arrival expressions (such as a single “salir”) gave recognition problems. We decided to allow them only in confirmations, and only when prefixed by a negating word.

Date expressions are described in section 3.3.3 (page 51).

These are the expressions accepted when expecting a time (the “don’t know” answer is needed because of the special dialogue structure, there is no reason why a time question should normally be answered with this):

- (don’t know)
- [marchar I irme I salir] **time-expr**
- Llegar **time-expr**
- [marchar I irme I salir] **time-expr**
- Llegar **time-expr**
- [time-expr ... [marchar I irme I salir]]
- [time-expr ... llegdar]
- time-expr
- el más rápido
However we found out the short departure/arrival expressions (such as a single “salir”) gave recognition problems. We decided to allow them only in confirmations, and only when prefixed by a negating word.

Time expressions are described in section 3.3.3 (page 52).

Expressions accepted when expecting an AM/PM expression:

- [por | de] la madrugada  
  - At dawn.
- [por | de] la mañana  
  - In the morning.
- [por | de] el mediodía  
  - Around noon.
- [por | de] la tarde  
  - In the afternoon or evening.
- [por | de] la noche  
  - In the evening or at night.

Expressions accepted when expecting a reply to a yes/no question (these are also used elsewhere, and are marked with ⟨yes/no⟩):

- sí  
  - Yes.
- sí que lo quiero  
  - Yes, I want that.
- bueno  
  - Yes.
- bueno, de acuerdo  
  - OK, that's all-right.
- sí sí  
  - Yes!
- claro  
  - Of course.
- claro que sí  
  - Of course.
- no  
  - No.
- no me interesa  
  - No, I'm not interested.
- no hace falta  
  - No, I don't need that.
- bueno, no  
  - Well, no.
- claro que no  
  - Of course not.

Expressions accepted when expecting a reduction criterion:

- no  
  - No.
- no me interesa  
  - I'm not interested.
- horario  
  - Schedule.
- según la hora  
  - Time.
- según cuándo sale el tren  
  - When it leaves.
- cuándo llega  
  - When it arrives.
- servicios  
  - Services.
- servicios que ofrece  
  - Offered services.
- su tipo  
  - Train type.
- el tipo de tren  
  - Train type.
- según la clase  
  - Train type.
- clase de tren  
  - Train type.
- barato  
  - Cheap.
3.3. INPUT SENTENCES

«el tren más económico»

*Cheapest train.*

Expressions accepted when expecting a train type:

«(don’t know)»

*I don’t know.*

«train-type»

*Train-type. (there is a list of train types in [4])*

«Quisiera viajar en train-type»

*I’d like to travel on a Train-type.*

«Quiero coger un train-type»

*I’d like to take a Train-type.*

Expressions accepted when expecting a train service:

«(don’t know)»

*I don’t know.*

«train-service»

*Train-service. (there is a list of train services in [4])*

«Un tren con train-service»

*A train with train-service.*

«Quiero viajar en un tren que tenga train-service»

*I’d like to travel on a train with train-service.*

Expressions accepted as selections of single trains in a list:

«(yes/no)»

*Yes/no.*

«(don’t know)»

*I don’t know.*

«el primero»

*On the first train.*

«el segundo»

*On the second train.*

«el tercero»

*On the third train.*

«el último»

*On the last train.*

«el del medio»

*On the middle (second) train.*

«el próximo tren»

*Next.*

«el tren siguiente»

*Next.*

«el de después»

*Next.*

«anterior»

*Previous.*

«el de antes»

*Previous.*

«el tren que llega antes»

*The earliest-arriving train.*

«que llegue antes»

*The earliest-arriving train.*

«que salga antes»

*The earliest-leaving train.*

Expressions accepted when expecting a kind of trip:

«no»

*No.*

«mismo»

*The same trip.*

«el mismo»

*The same trip.*

«el mismo viaje»

*The same trip.*
la vuelta
el de vuelta
para el viaje de vuelta
otro
sobre otro viaje

The return trip.
The return trip.
The return trip.
Another trip.
Another trip.

All top-level grammars named with `@__exp__x` handle answers to explicit questions about the same topic that `@__x` would normally handle. The computer asks the user to confirm that it understood her correctly.

Expected answers are “yes”, “no”, “no” followed by a correction, or simply the correction without any “no” before it:

> (yes/no) Yes/no.
> no, ...  x  No, x.
> ...  x  x.

for explicit confirmations of times, also AM/PM expressions are allowed as `x`.

Implicit questions are a way of telling the user what the computer understood, and giving the user a chance to correct the computer, while asking the user a normal question about something else.

All top-level grammars named with `@__simp__x` handle answers to implicit questions joined with normal questions that `@__x` would handle.

The user may answer the full question (which implies that she agrees with what the computer was trying to confirm), or say that the computer understood her incorrectly. It would be nice to allow the user not only to signal a recognition error, but also to correct it, as in explicit questions. However this would yield very large grammars, and the benefit of having restricted dialogues would be partly lost:

> (yes/no) Yes/no.
> no  ... No, ...
> perdón  ... Sorry, ...
> he dicho  ...  I said ...
> quiero decir  ...  I mean, ...
>  ...  x  x
> sí, ...  x  Yes, x

Note that “no ... ” “perdón ... ”, “he dicho ... ” and “quiero decir ... ” introduce a negative confirmation with a correction. The negation is detected but the corrected data is ignored.
3.3. INPUT SENTENCES

3.3.3 Date and time expressions

The expressions that the computer can recognize for dates and times may be rather complex. We give here a list of these recognized expressions. The actual recognizer grammar may be more complex than this (for example, the “…” are replaced with actual sub-grammars), so this list should be taken as an informal reference. The formal reference is the grammar file gram_recon/gram-bnf.txt (see C.1.1 on page 207).

For expressions involving word classes (for example, month names), we give an example using only one word in the class.

The recognized expressions for dates and date intervals are:

- [el] lunes próximo  
  - next Monday
- [el] lunes  
  - Monday
- lunes que viene  
  - next Monday
- este lunes  
  - this Monday
- [el] próximo lunes  
  - next Monday
- [el] fin de semana próximo  
  - next weekend
- [el] fin de semana que viene  
  - next weekend
- [el] próximo fin de semana  
  - next weekend
- [el] fin de semana  
  - the weekend
- este fin de semana  
  - this weekend
- [el] próximo día veintisiete  
  - next 27th
- [el] próximo veintisiete  
  - next 27th
- [el] [dia] 27th  
  - (the) 27th
- [el] [dia] veintisiete de enero  
  - January 27th
- [el] [dia] veintisiete del uno  
  - 01-27

- Navidad  
  - Christmas
- Año Nuevo  
  - New Year’s Day
- Nochebuena  
  - Christmas Eve
- Nochevieja  
  - New Year’s Eve
- Reyes  
  - January 6th
- pasado mañana  
  - the day after tomorrow
- mañana pasado  
  - the day after tomorrow
- mañana  
  - tomorrow
- hoy  
  - today
- dentro de tres dia[s]  
  - in 3 days
- un lunes  
  - any Monday
- los lunes  
  - any Monday
- un fin de semana  
  - any weekend
- los fines de semana  
  - any weekend
- cualquier lunes  
  - any Monday
Chapter 3. Designing the Dialogue Application

- [un] lunes cualquiera
- cualquier fin de semana
- [un] fin de semana cualquiera
- laborable
- festivo
- un día festivo
- de cada día
- entre semana
- una fecha entre semana
- de fiesta
- el día siguiente
- el día anterior
- el día antes
- el día después
- el día antes de
- el lunes antes de
- el día después de
- el lunes después de
- el segundo martes de enero
- el segundo fin de semana de enero

Any Monday
Any weekend
Any weekend
Workday
Holiday
Holiday
Workday
Workday
Holiday
The next day
The previous day
The day before
The day after
The day before...
The first Monday before...
The day after...
The first Monday after...
The second Tuesday in January

The recognized expressions for times and time intervals are:

- hacia ...
- de ... a ...
- entre ... y ...
- a partir de ...
- antes de ...
- después de ...
- antes del mediodía
- después del mediodía
- antes de la noche
- de madrugada
- por la mañana
- hacia media mañana
- al mediodía
- por la tarde
- a media tarde
- por la noche
- temprano

Around ...
From ... to ...
From ... to ...
After ...
Before ...
After ...
Before noon
After noon
Before night
At dawn
In the morning
(The central hours of the morning)
At noon
In the afternoon
(The central hours of the afternoon)
At night
Early
3.3. **INPUT SENTENCES**

- de día  
  - a primera hora  
  - hacia primera hora de la mañana  
  - a primera hora de la tarde  
  - a primera hora de la noche  
  - hacia última hora  
  - a última hora de la mañana  
  - a última hora de la tarde  
  - a última hora de la noche  
  
  - las trece  
  - las trece horas  
  - las dos  
  - las dos y cuarto  
  - las dos de la tarde  
  - las dos y cuarto de la tarde  
  - las trece y veinticinco  
  - las trece horas y veinticinco minutos  
  - ahora  
  - dentro de tres horas  

- during the day  
  - early in the day  
  - early morning  
  - early afternoon  
  - early in the night  
  - late  
  - late morning  
  - late afternoon  
  - late night  
  - 13:00  
  - two o’clock  
  - quarter past two  
  - two o’clock in the afternoon  
  - quarter past two in the afternoon  
  - 13:25  
  - now  
  - in 3 hours
Chapter 4

The Basurde[lite] modules

These modules (the programs, not the application definition files) have been
designed and implemented with the Basurde[lite] dialogue application in mind,
but also with the intention that they be flexible, powerful and extensible enough
to be useful for similar dialogue applications designed by other application
developers.

4.1 Speech recognition

The speech recognition module is not part of this project [10]. However, since
some of the data files controlling it are, a brief explanation of how the module
works is needed in order to understand these files.

This program has an internal model of the language it should accept. This
model is conceptually hierarchic, and structured as a directed weighted graph;
it is also possible to view the model as a non-hierarchic graph. At the top level,
a graph for a (possibly probabilistic) finite state automaton (FSA) describes
the accepted language using words as terminals. At an intermediate level, a
dictionary maps these words to sequences of phonetic units. Finally, at the
lowest level, we have graphs describing Hidden Markov Models (HMMs) [40]
for each phonetic unit. We can give the language model using even more
intermediate levels.

In addition, we have HMMs for whole words (“yes”, “no” and the ten digits).
The dictionary maps each of these words into one HMM. Having specially-
trained models for these words greatly increases the recognition success, espe-
cially for short sentences, or words that are easily confused (such as yes “sí”,
and seis “seis”).

Apart from this whole-word models, any kind of phonetic unit may be used.
For this project we use demiphones [32] for normal words, and phonemes for
a special catch-all word, the filler, which models any Spanish word and is
used to absorb unforeseen (and meaningless to us) expressions in the middle of sentences.
The graph has an initial and a final node. It is also possible to have multiple top-level finite state automata, and keep only one active for a given recognition run.
The weight on a transition going out of a node defines, through some transformation, the probability of that transition from that node. Based on a sequence of input vectors representing the input audio signal, the program decides which words the user said by finding, using Viterbi's algorithm, a path in the graph from the initial to the final node, which has maximum probability. That is, from an observation $O$, it finds a path $S$ such that

$$S = \arg \max_{S_i} p(S_i|O).$$

This is known as APP (a posteriori probability).
The program can then give the list of words (or phonetic units, or terminals at any other level) that generate $S$.

For Basurde[lite], we need to provide a language model to the speech recognizer. Phonetic models for Catalan and Spanish were already available from the research group. A program that transcribes any Spanish or Catalan word into demiphones is also available. So we needed to provide the grammars that have words as terminals, and transform these grammars and the list of words into the file formats that the speech recognizer uses.

See 5.1 (page 81) for more information about the language model we used in Basurde[lite].

### 4.2 Natural language understanding

The function of the natural language understanding module (or NLU) is to process sequences of words and recognition confidences given by the speech recognition module for a user utterance, produce Basurde frames that encode the meaning of that utterance and send them to the dialogue manager (see figure 4.1).

![Figure 4.1: Converting word sequences to frames](image)

So, basically, we need to interpret the input words, extract their meaning into some kind of internal representation, and produce frames for this internal repre-
4.2. NATURAL LANGUAGE UNDERSTANDING

sentation. Usually, such a translation task (as in a computer language compiler or translator) would be handled by a deterministic parser.

However, with natural language understanding we cannot do exactly that. The reason is that preparing a grammar for even a subset of a human language is a very difficult task and, in addition, even if the recognition process is completely correct, people do not talk in a grammatically correct way. Out-of-grammar events include doubting in the middle of the sentence, grammatically invalid word and concept ordering (following a sequence of thought rather than the language's normal ordering), correcting information given previously in the same utterance, and others. What is more, we have no guarantee that the words that the NLU module gets are really what the user said—the recognition module is not perfect. So a traditional parser (for a computer language) is not a good solution.

4.2.1 Proposed solutions

Many authors propose solutions involving some kind of partial parsing: We do not try to understand the whole user utterance. Instead we try to locate some meaning-rich portions which we know how to interpret, and discard the rest. The discarded part will usually be filler or connecting words or recognition errors. However we risk discarding important information which the partial parsers are not prepared to handle.

One technique is word spotting, where we scan the input word sequence for some pre-determined key words, and we apply some rules (typically stored in a table) to obtain the frame representation given the set of detected key words and possibly their ordering or other relations.

At a higher level, in phrase spotting [11] we look not for single words but for whole (short) syntactic and semantic phrases in a sentence.

Other proposed solutions are:

1. Using full deterministic parsing, with a grammar adapted to the special phenomena that are encountered in casual speech. This approach has the problem that it is a complex task to generate a grammar which can really handle all (or most) phenomena found in spoken language. On the other hand, there exist well-established tools (for example, yacc) that generate efficient parsers for a general enough class of grammars, when full parsing is used [2].

The biggest problem of all, however, is that not all grammars can be converted into a form which is usable by these parse-generation tools. It would be very confusing to an application developer to find an error in this step because her grammar was not of the right kind.
2. Using lookup tables to directly get a frame translation from an input word sequence. While a translator using this method may be extremely fast, these tables have to be quite large to take into account all the variability of human speech.

Our input language is finite, so another approach would be to generate all the elements of this language (all valid input strings), and assign a frame (or frame list) to each of them. Generating the language elements is easy (actually, we have done precisely this, but only partially, as a step to creating the language models for the speech recognizer), but there are so many of them that we would need some kind of interpreter in order to obtain their meanings. Of course, once we have this interpreter, we have a valid string-to-frame translator and we do not need a lookup-based translator other than for the extra speed.

We could follow an intermediate approach and obtain the corresponding frames by hand, assisted by a program. This program would be simpler than the full parser needed for the full-automatic approach. This has the disadvantage that this program would be too specific for this dialogue application. In addition, some grammars may be too large even for this computer-assisted processing. Since we did not have serious problems with the approach we chose, and it was reasonably fast, we did not look more into this option.

3. Some kind of trained parser, using probabilistic learning such as HMMs or a similar technique. This involves training a system by giving it a large number of pairs of input word sequences and their corresponding frames. While probabilistic learning is a well-studied subject, we did not have the time or resources to build the training data. For example, [19] uses example-based learning.

4. More elaborate techniques involve morphological tagging and full syntactic/semantic analysis in an attempt to detect the meaning of each word and phrase. These may include anaphora resolution ("cross-references") and a large knowledge base (information on the application domain).

This option is too complex for this simple machine-driven system.

4.2.2 The choice for Basurde[lite]

We decided to use a combination of two techniques, word and phrase spotting, together with a bottom-up parser as a method for finding the correct output for a given combination of spotted words and phrases. The parser operates in a mixed deterministic/non-deterministic mode.

Parsing a string for a given language, that is, deciding whether the string belongs to the language or not, and at the same time extracting relevant structural
information about the string in preparation for some kind of translation or interpretation, is a process that is, in the general case, time-exponential to the length of the string [29]. Informally stated, this means that if \( n \) is the number of words in the string, it takes on the order of \( k^n \) computation steps. For usual values of \( k \) and \( n \), this makes it very slow. In figure 4.2 we show a part of an example parse of an arrival time; we can see that even for such a simple string and rules, there are many possible intermediate steps.

![Diagram](image)

**Figure 4.2:** Part of an example parsing of an arrival time

There are some interesting grammars for which efficient parsers exist, for example, for the \( LL(1) \) and \( LALR(1) \) classes, part of the class of context-free grammars. However, we thought that limiting ourselves to context-free grammars might give some problems: context-free grammars can describe most (if not all) of the current computer languages, but we will not be dealing with computer languages but with human language. We decided that we would implement a parser for an unrestricted grammar.

**Extensions to the non-deterministic parser**

Our first attempt was implementing a non-deterministic-only parser that worked by, at each parse step, non-deterministically applying, in reverse direction, all rules that could have yielded the current string. In this way, we would successively reduce the length of that string. Each non-deterministic parsing thread stops when no rule can be applied to the current string. The parsing process ends when all parsing threads have stopped. At each step, we store all
intermediate strings.

When the parse ends, we search all stored intermediate strings for symbols marked as "goals" in the configuration file (these are analogous to the start symbol in a grammar). Conflicts are resolved using a priority scheme. Finally, one of these strings is selected as the parse, and a goal rule is used to translate the string into a frame or a frame list.

This straightforward approach has, at least, two problems. First, the parsing is extremely slow because of the exponential behavior: at each parsing step, potentially many reduction rules can be applied. The system has to keep track of all possible reduced strings; at the next parsing step, the reduction rules will be applied not to one string, but to all reduced strings from the previous step. Second, the parsing process may never end if there are loops in the reverse grammar graph: for example, a rule set such as \( S \rightarrow S, S \rightarrow aSa \) creates this problem. Fortunately, our grammars do not have this problem; an extension which will be commented later would help solve it, should we need to use such grammars.

We incorporated extensions to the basic parser and grammar descriptions to help us solve these problems (see B.4.4 on page 171):

We extended the non-deterministic parse steps with deterministic parsing and tagging: First, we provide a mechanism to group adjacent words into a single element, or to attach tags to classes of single words and later process the input string according to these tags, not the actual word. This tagging and word joining is made only at the beginning of the parse process. It is the "spotting" phase of a word-spotting or phrase-spotting technique.

We also add "deterministic" reduction rules. These rules are similar in expressiveness to standard, non-deterministic reduction rules, but they are applied deterministically: At each parse step, before applying all possible non-deterministic reduction rules to the current strings, all valid deterministic rules are applied. However, they are applied sequentially rather than in parallel like non-deterministic rules. The output of a reduction is used as the input string for the next reduction. The order of definition gives the order of application for deterministic rules. Figure 4.3 shows how changing just one rule from nondeterministic to deterministic mode simplifies the parse space of the example given in figure 4.2: here we must join two adjacent numbers into a time, whenever possible.

Figure 4.4 shows this mixed parse process graphically.

This added determinism helps limit the exponential growth of the search space. The system still has, of course, exponential behavior, but we manage to reduce the constant factors (the \( k \) in \( k^n \)) enough to make it usable.
Figure 4.3: An example semi-deterministic parsing of an arrival time

To make reduction rules simpler to express we added conditions. These are algorithmic tests, given in Tcl, which are performed over a string before deciding if a rule can be applied to the string. A string must match the pattern given by the rule, and also pass this test. As a side effect, we can use this feature to solve the second problem stated above: by keeping a counter attached to one of the looping symbols we can stop the reduction for that thread if the counter gets over a certain threshold.

Accepted languages

The parser accepts reduction rules like

\[ a_0 \ldots a_n X_0 \ldots X_w b_0 \ldots b_m \rightarrow a_0 \ldots a_n Y b_0 \ldots b_m \]

where \(a_i\) and \(b_i\) are context symbols, and \(Y\) is the symbol that the string of non-terminals \(X_0 \ldots X_w\) gets reduced into. This reduction rule, of course, corresponds to production rules like

\[ a_0 \ldots a_n Y b_0 \ldots b_m \rightarrow a_0 \ldots a_n X_0 \ldots X_w b_0 \ldots b_m. \]

While this is not the canonical rule form for unrestricted grammars, it can be shown [29, page 232] that any unrestricted grammar can be converted into one with rules of this kind (with only one symbol replacing the non-context non-terminals).

The other parser features, such as tagging, deterministic rules, or conditions, simply make rule descriptions easier to write, and make the parse process faster,
input string

apply
rule 0
rule 1
...
rule n

tagging
rules

parallelize for each string

apply
rule 0
rule 1
...
rule n

deterministic
rules

non-deterministic
rules

apply
rule 0
rule 1
...
rule n

collect all reduced strings

any?

no

collect all intermediate strings

select “best” string

translate

frame

Figure 4.4: How the NLU parser works
but of course do not extend the class of languages that the parser is able to
process, which is that of recursively enumerable languages\(^1\).

It turns out that, in the end, we need to prepare a set of reduction rules. We
have, in principle, no guarantee that the language accepted by the NLU module
is the same as the language generated (or accepted) by the speech recognizer.
For the files we used for the Basurde[lite] dialogue application, we actually
know that the NLU language is a strict superset of the recognized language.
Therefore, all strings which the recognizer can output are accepted by the NLU
module\(^2\).

That the NLU language is a superset of the recognizer language, and not the
same set, has no negative consequences, and does not affect the behavior of the
whole Basurde[lite] system.

**Recognition confidence**

The NLU module includes a section that calculates an aggregate recognition
confidence value for the whole translated frame list, based on the individual
word recognition confidence values from the input string: The aggregate value
is a weighted average of the word values. Words that have been replaced by
reduction or tagging rules count at 100%. Words that are used as context of
reduction rules, but are not replaced by a non-terminal, count at 50%. Unused
words are not included in the average.

We could probably refine this procedure. However, the actual speech recognizer
does not provide confidence values, so we cannot conduct evaluation tests to find
better aggregation methods. The Basurde[lite] system can function nonetheless,
because we give each word a default confidence value of 1 (the maximum value).

After translating a string, the NLU module decides, based on two confidence
thresholds and the aggregate confidence value, whether it should signal a recogni-
tion error, and of which kind (major or minor). The aggregate confidence
value is made available to the Tcl communications layer, but presently dis-
carded because the Basurde protocol cannot use it.

**Internal representation of parse structure**

As we parse a string, we must somehow record the syntactical information that
we obtain, so that we can later translate the string into frames. Or we can

---

\(^1\)This is not strictly true, since the NLU parser is not capable of handling the empty string.
However, we can modify the Tcl communications loop to handle this special case outside of
the parser.

\(^2\)Actually, the recognizer can issue an empty string in some cases of recognition failure.
An empty string is not accepted by the NLU module, which generates a parse error. This is
the correct "parse" for a recognition failure.
translate the string right as we get enough structural information, but then we must store the partial translation somewhere. After all, our goal is to translate input strings, not to check whether they are valid or not.

We solve this problem using what we call yield strings (see B.4.3 on page 170). We can attach arbitrary Tcl strings to elements (either the original words or the non-terminals we replace them with) in the parsed string. Reduction rules state what this string should be, and we can use arbitrary Tcl program fragments to calculate them. Information about the elements that are used in matching the current reduction rule are available to the program that generates yield strings. Trees and other kinds of structures can be stored as “Tcl strings”.

The application developer is free to use this facility as she likes. For the Basurde[lite] dialogue application, we store information as key-value pairs as we obtain that information (with, rarely, a hierarchic structure). The goal rules use this information to build the resulting frame list. Figure 4.5 show a hypothetical example of this process.

![Diagram](image)

Figure 4.5: Temporarily storing structural information about the parsed string
4.3 Dialogue manager

The dialogue manager is the module that keeps track of the whole conversation, decides what the system should say to the user, and processes the user’s reply. It is the only module which must keep some state across dialogue turns: the NLU and ORG modules can work, in most cases, without any knowledge of what happened in previous dialogue turns for this same dialogue; the dialogue manager, however, must remember what the user said so that it knows where in the conversation it is. In particular, it should, at least:

1. Remember the information that the user gave to the system (desired departure date, desired train type, ...),

2. know at which point in the conversation we are in (asking for the arrival place, telling about a single train, ... ),

3. keep track of, handle, and correct possible recognition errors or user-initiated corrections,

An extended finite state automaton, like the one presented in 3.1.5 (page 31), is a natural representation for dialogues for the purpose of keeping track of the current situation. Handling errors (and confirming user information that the system is unsure of) must be represented in other ways.

The division of work among the NLU, ORG, and dialogue manager modules is not so much well defined. Some authors shift more work to the NLU or the ORG, but for the Basurde[lite] dialogue application we chose to divide tasks so that all history and all the state which is kept across dialogue turns is kept in the dialogue manager, thus making the NLU and ORG largely memory-less\(^3\).

4.3.1 Proposed solutions

We have found many proposals for dialogue managers. Most are not mutually exclusive. As we will see, our choice will be a hybrid approach incorporating some of these ideas, plus some adaptations to our precise task.

1. In [47], a multi-layered approach is proposed: the top layer is concerned with the user’s and system’s goals; the second layer deals with distribution of attention and coherence of discourse; the third layer gives guidance to the user; the fourth layer handles confirmation and verification (information grounding); finally, the fifth layer handles the actual utterances.

\(^3\)Nothing prevents the application developer to use the Basurde[lite] NLU and ORG modules in a stateful way, but we do not do that in the dialogue application we developed. Making the ORG stateful may be interesting for, for example, make some sentences look different the first time they are asked than the next ones.
2. Most systems use some form of slot filling, in which the system internally represents the situation as a form with some empty slots. The dialogue manager tries to fill in the slots, so it can complete the task. There is usually a question associated to each slot; this question prompts the user to give the desired information. There may be constraints on slot values. An extension, hierarchical slot filling [47], organizes the information hierarchically (for example, a subtree with a “departure time” root might have “hour”, “minute” and “AM/PM” leaves), and chooses the right depth depending on the user’s behavior. On a similar trend [33] uses simpler questions if communication problems are found, and the VoiceXML language [12] allows questions to be rephrased if they are not answered in the first time.

3. Many methods exist for the actual control of the dialogue system. Scripting languages describing the dialogue flow [48] are commonly used. Finite state automata, graphs of dialogue objects or modules, and other state transition networks [5] directly implement the dialogue graphs used for describing some conversational interfaces. For simple applications using slot filling on a single form, the slot order within the form gives an implicit ordering for the dialogue.

4. Some simpler systems [19] connect a set of case-value pairs coming from user frames to a pre-processed answer, using lookup tables or a similar mechanism. This method is far too weak for an application like Basurde[lite].

5. Networks, trees and complex graphs of nodes can also be used. A question and its answer (or, using hierarchic methods, a full subnetwork) is associated with each node. In [35], they use a tree with nodes marked “current”, “unvisited”, “suspended” and “closed”; non-leaf nodes are closed automatically when a certain combination of their descendants become closed. The frame manager uses this tree to guide the dialogue.

Using a directed graph, with questions at nodes and user replies as arcs, makes applications easier to design and gives them a more predictable behavior. [15] uses similar states for nodes.

6. Other systems use a goal/subgoal mechanism, with a stack where pending goals are stored [14].

7. In [18] a very powerful and flexible event-driven mechanism is described. Unfortunately we cannot expect to implement anything similar using C++ with the limited time available to the Basurde[lite] project.

8. An interesting sequencing and grouping mechanism, the agenda, is described in [38]. This is a sequence of forms, dialogue acts, and applica-
4.3. DIALOGUE MANAGER

... tion actions (such as make a database query, confirm a form, ...). User requests trigger dynamic modifications in the agenda.

9. The dialogue manager described in [1] uses two mutually-interacting models, a finite state automaton for the machine, and a probabilistic FSA which models the user. The system works by estimating the hidden state of the user FSA, much in the way that Hidden Markov Models work. As far as Basurde[lite] is concerned, this method cannot be used because we lack the resources to build the training corpus needed to estimate the probabilities in the user model.

Summarizing, we can have a dialogue manager with a programmer-defined structure (we cannot use a learning-based system), using frames, frame hierarchies, graphs or agendas, all of which may be static or dynamic. We can use a single-level or a multi-level design.

4.3.2 The choice for Basurde[lite]

Basic concepts

Actually, we chose a hybrid approach taking all the interesting ideas which could be applied to our case, adapting them, and extending them with some new ones:

1. Tree-based slot filling: data given by the user, and information on how to ask questions to get that data, is stored in the nodes of a tree;

2. multi-layered dialogue manager: the dialogue manager is conceptually divided into a high-level manager, which handles the attentional state (what we are asking), and a low-level manager, which handles the grounding of information, confirmation, validation and checking. This division is not so clearly found at the implementation level;

3. dynamic tree growth: the application can dynamically add new nodes to the tree;

4. agenda-driven sequencing: dynamically-modifiable agendas, one per node, sequence the filling of descendant nodes and the execution of arbitrary application functions;

5. inter-node restrictions, dependences and inference: a set of special functions (acceptor functions, value-setting functions, value-access functions) may be used by the application developer to implement these.

In the following sections we briefly describe the behavior of the dialogue manager. See B.5 (page 181) for more detailed and precise information.
CHAPTER 4. THE BASURDE[LITE] MODULES

Task representation

All the information is stored in hierarchical form, as a tree, the “task tree” or “knowledge representation tree”. Each tree node contains a value slot, where we store pieces of information given by the user or inferred by the program. The program may attach additional information to a node, using the node’s node variables.

The dialogue starts by building a tree for holding the necessary information. The dialogue manager will use information stored in the tree to try to fill in each node. Although Tel is not an object-oriented language, we provide facilities to give the programmer an OOP feeling when building a task tree: for example, she may define parameterized templates (or classes) which can be instantiated in different points of the tree. This is interesting for reusable dialogue components such as “obtaining a time”. The application may dynamically extend the task tree in the middle of a dialogue by adding new subtrees and nodes to it.

Figure 4.6 shows a task tree. Each “pizza” subtree should probably be instantiated from a “pizza” template.

![Diagram of a task tree](image)

Figure 4.6: A task tree

Knowledge representation

We find three different kinds of nodes:

1. Container nodes act as containers for their children nodes, but do not add any additional information. For example, a “pasta dish” node might be a container for a “kind of noodle” node and a “sauce” node. The program, or the application definition files, provide a way to obtain the aggregate information from the parent node.

2. Leaf nodes contain atomic information and do not have any subtrees.
3. Some nodes have an intermediate behavior. For example a “time” node may dynamically grow an “AM/PM-ness” child if the user does not give the AM/PM-ness for the time.

In addition, the system supports the notion of a node’s data type. There is currently only one implemented data type, but nodes of different types are supposed to handle their data in different ways.

Not just a single value is stored in a value slot. A part of the full history of user input is saved, together with additional information that describes the state of stored user information:

1. The confirmation status is empty (the value slot is empty), inferred (this piece of data has been inferred by the system), recognized (the system got this piece of data from the user, but it has not been confirmed yet), confirmed (the user confirmed that this information is correct) or rejected (the user confirmed that this information is wrong) (see figure 4.7).

2. The recognition confidence, only for data with the recognized confirmation status, is related to the recognition confidence for the frame that gave this information.

![Figure 4.7: Confirmation status after a dialogue fragment](image)

The program provides functions to access the compact version of the data stored in the value slot. How this data is compacted depends on the node’s data type, but for example it might be removing all data elements that have been rejected, and using the most recent inferred, recognized or confirmed data.

**User interaction**

Each node which may be filled with information from the user also contains instructions on how to ask the user for it, and how to process its reply:
1. The frame (or frames) which should be sent to the output system (ORG) for a normal question, an implicit confirmation request, and an explicit confirmation request.

2. The control packets which should be sent to the input system (recognizer and NLU) to prepare them to receive the corresponding user reply.

3. Functions which should process input frames to fill in the slot (acceptor functions).

The dialogue manager keeps a record of which node has the current attention focus. When the manager receives an input frame, it will give it to this node’s acceptor function.

**Node activation states**

A node can be in several *activation states*: disabled, waiting, focused and closed (or complete). All nodes start in the disabled state, except the root node which starts in the focused state. The manager will ask questions to try to fill the value slots for nodes in the focused state.

The dialogue manager automatically changes the activation states: When it detects that information provided by a node is required, the node will be marked as waiting. Functions in parent waiting or focused nodes, coordinated by the high-level manager, tell which information provided in which sub-nodes is required.

The high-level dialogue manager scans through the task tree to find the innermost focused nodes, and tries to fill their value slot. When a focused node has its value slot filled, and the acceptor function agrees to, it is changed to the closed state. A focused node having all children in the complete state and having enough information to compute its own value, or having its own value slot filled, may be (depending on the actual settings for the node) changed to the closed state.

The application developer may also change the activation state of individual nodes by using certain commands in the application definition files.

The dialogue manager is supposed to ask questions about all focused nodes which have questions to be asked (nodes which act only as containers will be focused when one of their children is, but no question will be asked for the parent node). We cannot ask too many questions at a time, which is why we have separate *waiting* and *focused* states: The waiting state signals that the application wishes the dialogue manager to ask questions for a node. When the dialogue manager is not too busy making questions about other nodes, it will turn a waiting node into the focused state, and start asking about it. More precisely, when we have less than two focused nodes with questions to ask, the
dialogue manager will select a waiting node which is a child of a focused node, and mark it focused. It will use node agendas to determine the correct order. Each node also has a node priority, set by the dialogue manager, which helps decide which nodes should be asked about if at any time there are more than two candidates (we cannot ask more than two questions at a time, a full question and an implicit confirmation). This situation can occur if the application program manually changes activation states, for example if it decides, in the middle of a conversation, that it needs to ask an already answered question again.

Node agendas

Each node has an agenda, which is a queue of “things to do”. The application program can add new items at the end of this queue (or, for special situations, at the beginning). This is usually done in response to certain user selections or replies.

The agenda can also be cleared, and each node can define a starting agenda, a set of items which will be entered into the agenda every time the node goes to the waiting state.

Several kinds of item can go into an agenda:

1. A fill command, which is a request to the dialogue manager to obtain the information for a given node. This sets the node priority and the activation state.

2. A prio command, which only sets the node priority, in preparation for issuing a fill command for that node later on.

3. A flush command. No further items from the agenda will be executed until all children of this node are closed. We use this to force all pending confirmation requests to be resolved before we go on with a “yes/no” question, or to force all confirmations before a database query.

4. A call to any Tcl procedure, with any arguments. That is the way that procedures that do the actual “work”, such as making a reservation, or querying a timetable database, are scheduled.

The low-level manager

The high-level manager keeps track of the current focused node. The low-level manager receives from the high-level manager requests to fill in the value slot in this node. It calls the adequate question-forming functions, interprets the results (filtering out confirmation replies and meta-communication), and handles confirmation requests.
We want to be able to issue an implicit confirmation request for a node and a full question for another node (probably its next sibling). To do this, we must allow more than one leaf node to be focused, and specify which kind of focus each node is in: parent (this node is focused, but we should ask its descendant nodes only, not the node itself), ask (ask a full question for this node) and confirm (ask a confirmation question for this node; explicit or implicit depending on overall dialogue state).

The dialogue manager can handle two nodes which are focused and not in the parent focus state. Since only a finite number of focus-kinds are allowed, simple rules can be used so that the manager is able to coordinate several requests (one for each focused node). These rules are implemented on the C++ side.

Node functions

Some special functions are defined for each node. Default functions are provided in case the application developer prefers not to give explicit functions of these kinds.

1. An acceptor function, which will be passed a single frame when the user replies to a question. The function may react in four ways:

   (a) It may accept the frame. In this case, it should probably change the value slot for its node through the value-changing function.

   (b) It may reject the frame, because it is not the kind of frames that belong to this node. In this case, the frame will be passed to the acceptor function of that node’s parent, and so on. It is an error if the root node rejects a frame. Most parent acceptors will pass the frame to their children, and let each of them decide if they take the frame or pass it to the next child. The specific behavior of acceptor functions is described in B.5.6 (page 188).

   (c) It may accept the frame (as in the first case), but tell the dialogue manager that other nodes (in particular its parent node) may also be interested in that frame and should also receive it.

   (d) It may make some changes to the task tree activation state and ask for the frame to be reissued. The dialogue manager will evaluate the new situation created by the activation state changes and determine again which nodes should receive input frames. It will then send the frame again to that new receiving node.

If the user generates multiple frames, this function will be called multiple times, one per frame; only after a node has seen all frames will the remaining frames be passed to the parent node. This function can keep state across calls by attaching the state to node variables for its node.
Constraints on node values may be implemented by this function, by rejecting a frame depending on its contents.

The default acceptor function rejects all frames.

2. A value-changing function, which should set the value for this node's value slot, probably after some pre-processing. This will be called by acceptor functions from this node, by value-changing functions for other nodes, and other functions. Not only the value, but also the confirmation status and other associated data is passed to this function.

The normal Tcl functions for setting value slots call this function, if available. Therefore, when setting the value for the current node's value slot, this function should use the special functions that bypass value-changing functions.

This function may also call the value-changing function for other nodes, and maybe change their activation status. This is how inter-node dependencies may be implemented. However, the application developer must take care of not creating circular references. It may also modify node agendas.

This function should change the activation state for its node. For example, if the new value for this node's value slot has the confirmed confirmation status, it may set its node to the closed state. If you want to disable confirmations for a node (directly accept the first user reply), this function should close the node when it receives values also in the recognized confirmation status.

The default value-setting function sets the value and confirmation status for this node's value slot, and closes the node if the value has a confirmed confirmation status.

3. A value-access (or compaction) function is used to obtain a single value from the value slot of this node, taking into account all the saved history of values and confirmation status, for example, ignoring rejected values, joining compatible values with the same confirmation status, ...

There are several default value-access functions. One, designed for leaf nodes, returns the most recent confirmed, recognized or inferred value. For nodes with descendant nodes, there is a function that returns, from all the children's values, that which has a highest ("most-confirmed") confirmation status, and another that returns all the children's values.

**Accessor functions**

Functions are provided to access any node, given an absolute path or a path relative to the current node. These functions
CHAPTER 4. THE BASURDE[LITE] MODULES

- access and modify a value slot.
- access and modify node variables.
- access and modify activation states.
- access and modify focus kinds.
- access and modify confirmation statuses.
- do dynamic instantiation and appending of subtrees.

4.3.3 An example

To clarify all this, we present a simple example involving a dialogue subtree for storing a date, either a departure date or arrival date. It will ask for a departure date, but is also ready to accept an arrival date. It only needs to have one of them, not both. It is composed of three nodes: a container node, a node for holding the departure date, and a node for holding the arrival date.

The subtree is part of a larger task tree, and the example starts with this subtree in the disabled state. Let’s assume that there are no nodes in the ask or confirm focus kinds in the hidden part of the tree.

![Diagram](image)

Figure 4.8: Dialogue manager example: Initial state. After focusing on the “date” node.

Figure 4.8 (left) shows the initial state of the subtree. All three sub-nodes are disabled. The container node has its initial agenda ready (agendas are shown in light blue).

After the “date” sub-node is brought into the focused state, its initial agenda will be executed. This will set the “departure” node to the waiting state, and will give priority values for both nodes (the “arrival” node gets it, too, because of the prio agenda command). The flush command cannot be executed, because
the node has open children (we really do not need a flush command for this example, but we use it to show its behavior). Figure 4.8 (right) shows the state of the subtree at this point. Activation states are shown in yellow; priority values are shown in magenta.

Figure 4.9: Dialogue manager example: Asking for the departure date. We receive an arrival date.

The dialogue manager decides to focus on the “departure” node (there is no other choice, since the “arrival” node is not even in the waiting state). Since the “departure” node has not received any answer yet, it changes to the ask focus kind (focus kinds are shown in green). It outputs a frame which the ORG will turn into a question about the departure date. This is shown in figure 4.9 (left).

However, the user decides to reply with an arrival date instead of a departure date. Figure 4.9 (right) shows how the answer frame is sent to the “departure” node. This node rejects it, since it can only accept frames about departure dates. The frame is then rerouted to the parent node.

The “date” node has a special acceptor function (a multiplexing acceptor), which is designed to accept all the kinds of frames that its children can accept, and reroute them to the right child. Figure 4.10 (left) shows how this node analyses the frame, decides that the “arrival” node can handle it, and then closes the “departure” node and sets the “arrival” node to the focused activation
Figure 4.10: Dialogue manager example: “Departure date” node rejects frame. Frame is routed to the “arrival date” node.

state and ask focus kind. Finally, the application requests the dialogue manager to reissue the frame. Since the new focused node is the “arrival” node, the frame is sent to it.

In figure 4.10 (right) we see how the “arrival” node accepts the frame. It then sets gives this new value ("tomorrow") to its value slot (node values are shown in pink). The focus kind changes to confirm: now the dialogue manager has to try to confirm this information.

Figure 4.11: Dialogue manager example: Confirmation request for arrival date. Date is confirmed.

Figure 4.11 (left) shows how the manager issues an explicit confirmation request for the arrival date. It cannot issue an implicit confirmation request because
we have no more pending nodes in the tree.

The user replies to the confirmation request with a “yes”. This frame is automatically handled by the low-level dialogue manager — the application developer does not need to worry about them if the default confirmation behavior is good enough for her application. As shown in figure 4.11 (right), this turns the “arrival” node to the closed state.

![Diagram](image-url)

**Figure 4.12:** Dialogue manager example: The whole “date” subtree is closed.

Once the “arrival” node is closed, all children of “date” are closed. We can now execute the flush agenda command. The agenda becomes then empty. The “date” node changes to the closed state, too. Notice how we can access children values through the parent node.

## 4.4 Oral response generator

The function of the oral response generator module, or ORG, is to convert frames generated by the dialogue manager into Spanish text.

This is the simplest module of all three Basurde[lite] modules. The application definition file gives a list of translation rules. When the module receives a list of frames to be translated, it sequentially scans the list of translation rules looking for one or more rules that can be applied to the received frames. Applying the rules gives the output text.

This is the template-based approach, the most widely used. However, the fact that production and matching rules can be full Tcl sub-programs makes this version especially powerful. On the other hand, the method used to determine which rule matches the received frames is not very general. However, it has proved to be general enough to handle all of Basurde[lite].
Each rule contains a frame description list (see B.6.1 on page 196) which describes what frames it can be applied to. A single rule can be applied to more than one frame if its frame description list contains descriptions for more than one frame, and all the descriptions match the frames the rule is applied to.

A description for a single frame contains the frame type, which must exactly match that of the received frame, and descriptions for zero or more cases. The developer may state that a rule should match frames with exactly the defined cases, or that extra cases are allowed in the input frames.

A description for a case contains the case name, which must exactly match that of a corresponding case in the received frame, and a case value. The case value can be a literal string, which must exactly match the case value of the corresponding case, or a variable name. Case descriptions with variables match cases with the same case name and any case value. In the matching process, a variable with that name will be set to the real case value. This variable can be used during the generation of the output text. Also, a case description may be marked as "optional"; a received frame does not need to have a corresponding case (but if it does, its case value must match a corresponding literal case value).

Figure 4.13 shows an example of frame matching and translation.

**Figure 4.13: An example of ORG frame matching and translation**

The program will try to find rules that match the received frames, starting with the first received frame. If there is a tie, it will use the rule that matches more frames. If there is still a tie, it will use the rule that matches more cases. When a matching rule is found, the output for this rule will be calculated. All the matched frames will be removed and the program will try to find rules for
the remaining frames. Finally, all outputs will be concatenated and sent to the speech synthesizer.

The rules may specify their output text as either a string or a Tcl code fragment. Variable and command substitution will be performed on the string before taking it as output. For code fragments, they will be executed by the Tcl interpreter and their return value will be used as output text. During execution of the code fragment and substitutions on the string, some variables will be set to the case values that were matched by case descriptions with variable names.

As another example, here is a hypothetical frame description which might be used for telling a departure city to the user:

```
system_information (...
  departure-city: c
```

If the output string were

“The train leaves from $c.”

and the following frame were sent to the ORG module,

```
system_information (...
  departure-city: Barcelona
```

then this rule would be selected, and the output text for it would be

“The train leaves from Barcelona.”

More details on the behavior of the ORG module can be found at B.6 (page 196).
Chapter 5

Implementing the dialogue application

5.1 Language models for the speech recognizer

In this section we describe how we prepared the language models for the speech recognizer.

The language model is the description of what kind of things the user may say. By providing a language model to the speech recognizer we help it better understand the user, since we give hints on what words are most expected at a certain place in a sentence, or following certain words. For example, it is easy to confuse “hat” and “cat” as isolated words. However, we can tell the computer that after the words “I am wearing a”, it is far more possible to have a “hat” than a “cat”.

Instead of using the n-gram approach, where we would give the probability of finding each word given the previous n words, we use the grammar approach; We use a grammar (in this case, a regular grammar specified with a finite-state automaton) to define the set of all acceptable word sequences. The recognizer does not accept anything which is not in that set.

Compared to the probabilistic approach, this has the advantage that input to the NLU model is very much well defined. We can choose to recognize only those sequences that we know how to interpret in the later modules. On the other hand, we may fail to capture all reasonably valid user replies. Or worse, if the user says something which is not defined in the grammar, the recognizer may try to return the “closest sentence” found in the grammar, which may be completely unrelated to the original sentence.

We also use another feature: Instead of having just one big grammar, we have
many smaller grammars, one for each recognition state. Each grammar generates a specific kind of user replies (for example, a date, or a train type).

Having one different grammar for each kind of user reply, and having only one active grammar at a given time (depending on what the dialogue manager expects the user to say) instead of one big grammar that can understand everything, has the advantage that, if the user actually says something that fits in that grammar, recognition success rate will be higher, since the recognizer will have to deal with a smaller grammar. On the other hand, the reply is even more constrained.

We have the following components:

1. One grammar for each recognition state. Their terminals are Spanish words, Basurde[lite] identifiers (for places and some train types), second-level grammars each representing the same meaning (for example, we have a grammar that generates many possible ways of saying “yes”), and the special models for out-of-vocabulary sentences @oo v and optional filler words @f.

   There is a tool (abnf2fsa) that can produce, among other things, descriptions of FSAs in the right format, given a description in BNF.

   We also have a tool that transcribes to demiphones most Spanish and Catalan words (mar2dicc).

2. Hidden Markov Models for Spanish and Catalan demiphones, for Spanish phonemes, and for some Spanish words ("yes", "no" and the ten digits).

3. A computer-readable description of finite state automata (FSA) which accepts the @ov and @f symbols. The corresponding grammars have Spanish phonemes as terminals.

   These symbols are trash models (see 5.1.2 on page 85) and are used to capture unrecognized words.

We created the grammar file by hand, by analyzing all the constraints and information gathered in the design phase (see 3.3 on page 44). See the comments on the grammar file gram-bnf.txt (C.1.1 on page 207) for more information.

We also prepared a set of scripts (see C.1 on page 207) which automates the whole process of obtaining the language model files for the speech recognizer, given a grammar file and the exception lists for the phonetic transcription. Thanks to it we were able to easily and quickly fine-tune our language model as we performed the first complete tests.
5.1. LANGUAGE MODELS FOR THE SPEECH RECOGNIZER

5.1.1 Layers

The speech recognizer requires that its language model have a layered structure. In each layer we can have only one of the following:

1. A set of HMMs of any kind;
   
   (for example, we could give one HMM for the phoneme /θ/, one for /r/ and one for /i;/)

2. a dictionary which maps symbols (terminals of the next higher level in the language model) into sequences of symbols defined in the next lower level in the language model;
   
   (for example, this dictionary could contain the rule three → /θri:/, or rather three → /θ/ /r/ /i;/)

3. a set of finite state automata, having as terminals symbols defined in the next lower level. An FSA cannot use as terminals other FSAs in the same level.
   
   (for example, we could have an FSA for a grammar with this rule:

   \[ \text{number} \rightarrow \text{one|two|three|...} \] )

We decided to structure our language model in the way shown in figure 5.1 (page 83). The parts that we developed for Basunde[lite] are shown in light blue.

![Layered structure of the language model](image)

Figure 5.1: Layered structure of the language model

At the first (top) level, we put the FSAs for our grammars.
Since most terminals for our grammars are words it would seem logical, at first sight, that at the second level we should have the dictionary. However, some grammar terminals are not words: the \texttt{f} and \texttt{oov} terminals are implemented as FSAs. Therefore, due to restrictions in the speech recognizer, the second level must also be an FSA level.

We decided to take advantage of this "extra" FSA level to do the following: instead of fully expanding our top-level grammars and putting their FSAs at the top level, some non-terminals would be left unexpanded at this top level. We put their expansion in the second level. It is important that the terminals of these second-level grammars are words only (not \texttt{f} or \texttt{oov}). We then ask the recognizer to return (as the recognized string) terminals of top-level grammars. By grouping in a second-level grammar expressions with the same meaning, we get some simple parsing done at the recognizer level.

For example, we could have a top-level grammar with the rules

\[ \text{sentence} \rightarrow (\text{don't know} \mid \text{yes} \mid \text{no}) \ast \text{date} \]
\[ \text{date} \rightarrow \text{tomorrow} \mid \ldots \]

and a second-level grammar with

\[ \text{don't know} \rightarrow \text{I don't know} \mid \text{well} \mid \ldots \]
\[ \text{yes} \rightarrow \text{yes} \mid \text{of course} \mid \text{sure} \mid \ldots \]
\[ \text{no} \rightarrow \text{no} \mid \text{of course not} \mid \text{no way} \mid \ldots \]

among other rules, if the user said "I don't know, well, tomorrow of course" the speech recognizer might return "don't know don't know tomorrow yes", with already some processing done.

At this point we have, as terminals of the second level, words (from the second-level grammars) and phonemes (from the \texttt{f} and \texttt{oov} models). What about the words which are terminals of the first level? We needed to add to the second level a set of trivial FSAs that mapped words as terminals of the first level into words as terminals of the second level.

At the lowest level we put all the HMMs. We have three kinds of them: for Spanish and Catalan demiphones, some Spanish whole words and Spanish phonemes.

So, the only missing level is some translation from words (terminals of the second level) into sequences of HMMs. We use a dictionary (basically a lookup table) for this. The trivial translations from words to HMM models for words, and from phonemes (terminals of \texttt{f} and \texttt{oov}) to HMM models for phonemes can also be done using this dictionary.

We explore the grammars and prepare a list of all the words we find. We then use the available phonetic transcription tools to transcribe them to demiphones. Some words, which are incorrectly transcribed by the automatic tool,
are transcribed using an exception table. The trivial translations are finally added to the dictionary.

5.1.2 Comments on the trash models

The @f and the @ooov models are sometimes known as trash or catch-all models, because they are designed to “catch” out-of-vocabulary or unexpected words, and to let us ignore words at certain positions. For example, if we want to accept the following sentences

▷ To New York
▷ I go to New York
▷ I’d like to go to New York
▷ I’d like a ticket for New York, please
▷ May I have more information on trains to New York, please?

we could simply enter “@f New York @f” in the grammar file.

The FSAs that we use for these models are phoneme 3-grams, matching the trigram phoneme distribution for Spanish. In this sense, they accept words that “sound Spanish”.

However, we found two problems with this approach. First, it turned out that words that we wanted to detect were more easily recognized by these models than by models of the sequence of demiphones for the word. For example, if we had a grammar with only two members

▷ @f San Francisco @f
▷ Yes @f

(the second element may be used to catch a positive reply), when we tried to recognize “to San Francisco” the recognizer followed the second path and recognized “yes @f”. We tried to solve this problem by modifying the FSA files for the trash models so that there was a higher penalty for “using” them.

Also, the recognizer operated far more slowly when it was trying to recognize a trash model than a normal word. To reduce this problem, we expanded the grammar with “trash words” which were given in full. For example, the grammar explicitly allows “please”, as well as a trash model, at the end of sentences. In this way we expect to catch more “meaningless” words through explicit words in the grammar.
5.2 Frames on the input side

The following frames are defined at the interface between the natural language understanding module and the dialogue manager. They represent user utterances, or other data coming to the dialogue manager.

Most frames follow, at least partially, the Basurde standard [31, 44]. Those marked with † are not defined in Basurde, or are used in a more broad sense as that defined in Basurde; we needed to modify the Basurde frame standard because Basurde[lite] is a machine-driven system, whereas Basurde is a user-driven system, so they do not have exactly the same inputs and outputs. Some frames may actually not be used by Basurde[lite], but are there to get some compatibility with other Basurde modules.

There is a list of case values, case names, and frame types in 5.4 (page 100).

5.2.1 Comprehension failures (user made)

<table>
<thead>
<tr>
<th>Frame Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>† u_no-entendido</td>
<td>Asks the system to repeat its last interesting utterance.</td>
</tr>
<tr>
<td>† u_no-entendido</td>
<td>Asks the system to end the dialogue and transfer the call to an operator.</td>
</tr>
<tr>
<td>† u_no-entendido</td>
<td>When the user doesn’t start talking for a certain predefined amount of time when it’s her turn to talk, this frame is sent to the dialogue manager.</td>
</tr>
<tr>
<td>† u_no-entendido</td>
<td>This is a protection against severe misunderstandings between the user and the computer. At any time, the user can say “corrección”, and the system will restart the dialogue.</td>
</tr>
</tbody>
</table>

5.2.2 System failures

<table>
<thead>
<tr>
<th>Frame Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>† u_fallo-reconocimiento</td>
<td>The NLU module sends this frame when it cannot get the meaning of the user utterance it has received, or if the recognition confidence given by the speech recognizer falls below a certain threshold.</td>
</tr>
</tbody>
</table>
5.2. FRAMES ON THE INPUT SIDE

<table>
<thead>
<tr>
<th>Frame Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u_fallo-reconocimiento</td>
<td>If the NLU module is able to get the meaning of a user utterance, but the recognition confidence given by the speech recognizer falls below a certain threshold (higher than that for total recognition failures), a u_fallo-reconocimiento frame is given before the frame for the user utterance.</td>
</tr>
<tr>
<td>u_informacion</td>
<td>The user does not know which train type to choose.</td>
</tr>
<tr>
<td>u_informacion</td>
<td>The user does not know which train service to choose.</td>
</tr>
<tr>
<td>u_informacion</td>
<td>“Yes” answer to a yes/no question.</td>
</tr>
<tr>
<td>u_informacion</td>
<td>“No” answer to a yes/no question.</td>
</tr>
<tr>
<td>u_informacion</td>
<td>Specifies the AM/PM-ness of a time.</td>
</tr>
<tr>
<td>u_informacion</td>
<td>Gives a train type.</td>
</tr>
<tr>
<td>u_informacion</td>
<td>Gives a service offered on a train.</td>
</tr>
<tr>
<td>u_informacion</td>
<td>Selects a train by giving its position within a list (“first”, “second”, “last”, …).</td>
</tr>
<tr>
<td>u_informacion</td>
<td>Where w is “salida” or “llegada”, selects the train that leaves or arrives earliest.</td>
</tr>
<tr>
<td>u_informacion</td>
<td>Selects the fastest train.</td>
</tr>
<tr>
<td>u_informacion</td>
<td>Selects the cheapest train.</td>
</tr>
<tr>
<td>u_informacion</td>
<td>fecha-w: $d$</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>(where $w$ is “salida” or “llegada”)</td>
<td></td>
</tr>
<tr>
<td>Gives the departure or arrival date. This is used for single dates, date expressions referring to a single day.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>u_informacion</th>
<th>intervalo-fecha-w: $\tilde{d}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(where $w$ is “salida” or “llegada”)</td>
<td></td>
</tr>
<tr>
<td>Gives the departure or arrival date. This is used for date intervals, date expressions referring to a range of days.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>u_informacion</th>
<th>tipo-dia-w: $d_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where $d_t$ is “laborable” (workday) or “festivo” (holiday), and $w$ is “salida” or “llegada”, gives the departure or arrival date as a kind of day.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>u_informacion</th>
<th>$\dagger$ fecha-w: $?$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(where $w$ is “salida” or “llegada”)</td>
<td></td>
</tr>
<tr>
<td>The user gives this when correcting the date given by the system, when the date itself is correct but the fact that it is a departure or arrival date is wrong.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>u_informacion</th>
<th>hora-w: $t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where $w$ is “salida” or “llegada”, gives the departure or arrival time as a single time.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>u_informacion</th>
<th>intervalo-hora-w: $\tilde{t}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where $w$ is “salida” or “llegada”, gives the departure or arrival time as a time range.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>u_informacion</th>
<th>$\dagger$ hora-w: $?$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(where $w$ is “salida” or “llegada”)</td>
<td></td>
</tr>
<tr>
<td>The user gives this when correcting the single time given by the system, when the time itself is correct but the fact that it is a departure or arrival date is wrong.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>u_informacion</th>
<th>$\dagger$ intervalo-hora-w: $?$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(where $w$ is “salida” or “llegada”)</td>
<td></td>
</tr>
<tr>
<td>The user gives this when correcting the time interval given by the system, when the time itself is correct but the fact that it is a departure or arrival date is wrong.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>u_informacion</th>
<th>$\dagger$ tipo-viaje: mismo</th>
</tr>
</thead>
<tbody>
<tr>
<td>The user wants more information about other trains for this same trip.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>u_informacion</th>
<th>$\dagger$ tipo-viaje: vuelta</th>
</tr>
</thead>
<tbody>
<tr>
<td>The user wants information about trains for the return trip.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>u_informacion</th>
<th>$\dagger$ tipo-viaje: otro</th>
</tr>
</thead>
<tbody>
<tr>
<td>The user wants information about other trips.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>u_informacion</th>
<th>$\dagger$ criterio-reduccion: hora</th>
</tr>
</thead>
<tbody>
<tr>
<td>The user wants to select trains based on their schedule.</td>
<td></td>
</tr>
</tbody>
</table>
5.2. FRAMES ON THE INPUT SIDE

\[\textbf{u.informacion} \quad \text{\dag\ criterion-reduccion: tipo-tren}\]

The user wants to select trains based on their type.

\[\textbf{u.informacion} \quad \text{\dag\ criterion-reduccion: servicios}\]

The user wants to select trains based on the services offered on them.

5.2.4 Confirming system beliefs

To give a negative confirmation (a rejection) to an implicit confirmation request, we also use

\[\textbf{u.informacion} \quad \text{\dag\ si-no: no}\]

To give a positive confirmation to an implicit confirmation request, we use, only in the case where the confirmation is explicitly given by the user (with an explicit “yes”-like word),

\[\textbf{u.informacion} \quad \text{\dag\ si-no: si}\]

probably followed by the frames corresponding to the reply to the last full question made by the system (the one that went together with the implicit confirmation request).

If a positive confirmation is given implicitly, that is, by not giving a negative confirmation, no “si-no: si” frame is issued. However, there will usually be the frames corresponding to the reply to the last full question.

To give an explicit positive confirmation to an explicit confirmation request, we use

\[\textbf{u.informacion} \quad \text{\dag\ si-no: si}\]

It is not possible to give an implicit positive confirmation to an explicit confirmation request.

To give an implicit or explicit negative confirmation to an explicit confirmation request, we use

\[\textbf{u.informacion} \quad \text{\dag\ si-no: no}\]

followed, if the negative confirmation was also a correction (not only the user rejected the system’s belief but actually corrected it), by the frames corresponding to a normal user assertion of that information.
5.3 Frames on the output side

The following frames are defined at the interface between the dialogue manager and the oral response generator. Most frames follow, at least partially, the Basurde standard [31, 44]. Those marked with † are not defined in Basurde, or are used in a more broad sense as that defined in Basurde. Some frames may actually not be used by Basurde[lite], but are there to get some compatibility with other Basurde modules.

There is a list of case values, case names, and frame types in 5.4 (page 100).

We give in *italics* an example of how this frame could be translated to Spanish text. See the application definition files for the ORG module (C.5 on page 655) for the authoritative translation.

5.3.1 Comprehension failures

<table>
<thead>
<tr>
<th>Frame</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_no-entendido</td>
<td><em>Perdone, me he estropeado. Le paso con un operador.</em></td>
</tr>
<tr>
<td>† tipo: error-sistema</td>
<td>This is used when the system detects an internal error (not a recognition error).</td>
</tr>
<tr>
<td>s_no-entendido</td>
<td><em>Disculpe, pero necesito más información para poder ayudarle. Por favor, responda mis preguntas.</em></td>
</tr>
<tr>
<td>† tipo: falta-informacion</td>
<td>The system complains that the user refuses to answer important questions such as the departure and arrival places.</td>
</tr>
<tr>
<td>s_no-entendido</td>
<td><em>Lo siento, pero no consigo entenderle. Le paso con un operador, no cuelgue.</em></td>
</tr>
<tr>
<td>† tipo: cierre</td>
<td>This is used after a series of recognition failures, when the system decides that the dialogue is a unrecoverable failure.</td>
</tr>
<tr>
<td>s_no-entendido</td>
<td><em>¿Oiga?, ¿Hola? and similar expressions.</em></td>
</tr>
<tr>
<td>† tipo: timeout</td>
<td>This is used after a recognition timeout, when the user does not answer the system within a predefined time limit.</td>
</tr>
<tr>
<td>s_no-entendido</td>
<td><em>¿Cómo dice?, ¿Puede repetir? or No le he entendido, ¿puede repetir?</em></td>
</tr>
<tr>
<td>tipo: repeticion</td>
<td>The system asks the user to repeat what she just said, after a recognition failure.</td>
</tr>
</tbody>
</table>
5.3.2 Offers and politeness

<table>
<thead>
<tr>
<th>s_cortesia</th>
<th>(...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: entrada</td>
<td></td>
</tr>
</tbody>
</table>

_Buenos días, Le atiende el servicio de información sobre trenes regionales y de largo recorrido._

Initial greeting.

<table>
<thead>
<tr>
<th>s_cortesia</th>
<th>(...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: cierre</td>
<td></td>
</tr>
</tbody>
</table>

_Gracias por usar este servicio, y buen viaje._

Standard closing sentence for successful dialogues.

<table>
<thead>
<tr>
<th>s_cortesia</th>
<th>(...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: espera</td>
<td></td>
</tr>
</tbody>
</table>

_Un momento, por favor._

“One moment, please”. The system says this just before starting a database query. These queries take some time, and it is a good idea to tell the user to wait.

<table>
<thead>
<tr>
<th>s_cortesia</th>
<th>(...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: reposicion</td>
<td></td>
</tr>
</tbody>
</table>

_Repite._

“I repeat”. The system says this just before repeating its previous utterance, if the user asked it to repeat that. This is better than just repeating it, because we also tell the user that we recognized her command and that this is not new information.

<table>
<thead>
<tr>
<th>s_cortesia</th>
<th>(...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: oferta</td>
<td></td>
</tr>
<tr>
<td>concepto: instrucciones</td>
<td></td>
</tr>
</tbody>
</table>

_¿Quiere que le explique cómo funciona el sistema?_

The system offers the user a short tutorial on this system.

<table>
<thead>
<tr>
<th>s_cortesia</th>
<th>(...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: instrucciones</td>
<td></td>
</tr>
</tbody>
</table>

_Este sistema solamente puede proporcionar información sobre trenes regionales y de largo recorrido. No puede hacer reservas. Por favor, habla después de oir la señal, con frases cortas y simples. Para corregir, diga “no”. En el caso de errores acumulados, diga “corrección”, para volver a empezar._

This is a short tutorial on this system. We ask the user to talk only after a beep, and to use simple sentences. We also tell her how to correct the system; either by saying “no”, or saying “corrección”.

<table>
<thead>
<tr>
<th>s_cortesia</th>
<th>(...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: oferta</td>
<td></td>
</tr>
<tr>
<td>concepto: nueva-consulta</td>
<td></td>
</tr>
</tbody>
</table>

_¿Desea información sobre otros trenes? Por favor, responda sí o no._

“Would you like information on other trains? Please answer yes or no.”
| s_cortesia | (...) | ¿Quiere más información de este tren? Por favor, responda si o no.  
“Would you like more information on this train? Please answer yes or no.” |
| s_cortesia | (...) | ¿Quiere información del f(x) tren?  
where  
f(1) = primer  
f(2) = segundo  
f(3) = tercero  
f(1, 2) = primer o segundo  
f(1, 3) = primer o tercero  
f(2, 3) = segundo o tercero  
f(1, 2, 3) = primer, segundo o tercero  
f(+1) = siguiente  
f(-1) = anterior  
f(-1, +1) = anterior o siguiente  
asks the user if she wants information on the first, second, third, previous or next train in a list. |
| s_cortesia | (...) | ¿Desea el tren más rápido?  
Offers to look for the fastest train. |

### 5.3.3 Questions

| s_solicitud | (...) | ¿Qué tipo le interesa?  
Asks the user for a train type she might be interested in. |
| s_solicitud | (...) | ¿Qué servicio le interesa?  
Asks the user for a service (restaurant, movies, ...) she might be interested in. |
| s_solicitud | (...) | ¿Salir time(t) de am-name(t) o de pm-name(t)?  
Asks whether a given departure time should be taken as AM or PM. We use specific names for AM and PM parts of the day, suitable for that time, to make the system utterance more natural. For example, “Leave at six in the morning or in the afternoon?” |
### 5.3. FRAMES ON THE OUTPUT SIDE

<table>
<thead>
<tr>
<th>s_solicitud</th>
<th>descripción</th>
<th>pregunta</th>
<th>contex</th>
<th>notación</th>
<th>detección</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>s_solicitud (…)</strong></td>
<td>† concepto: hora-am-pm, hora-llegada: t</td>
<td>¿Llegar time(t) de am-name(t) o de pm-name(t)?</td>
<td>Asks about the AM/PM-ness of an arrival time.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>s_solicitud (…)</strong></td>
<td>† concepto: hora-am-pm, intervalo-hora-@salida: t</td>
<td>¿Salir time(t) de am-name(t) o de pm-name(t)?</td>
<td>Asks about the AM/PM-ness of a departure time interval.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>s_solicitud (…)</strong></td>
<td>† concepto: hora-am-pm, intervalo-hora-llegada: t</td>
<td>¿Llegar time(t) de am-name(t) o de pm-name(t)?</td>
<td>Asks about the AM/PM-ness of an arrival time interval.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>s_solicitud (…)</strong></td>
<td>concepto: ciudad-destino</td>
<td>¿Cuál es su destino?</td>
<td>Asks about the destination.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>s_solicitud (…)</strong></td>
<td>concepto: ciudad-origin</td>
<td>¿Desde dónde desea viajar?</td>
<td>Asks about the departure place.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>s_solicitud (…)</strong></td>
<td>concepto: fecha-salida</td>
<td>¿Qué día desea salir?</td>
<td>Asks about the departure date.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>s_solicitud (…)</strong></td>
<td>concepto: fecha-llegada</td>
<td>¿Qué día desea llegar?</td>
<td>Asks about the arrival date.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>s_solicitud (…)</strong></td>
<td>concepto: hora-salida</td>
<td>¿A qué hora desea salir?</td>
<td>Asks about the departure time.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>s_solicitud (…)</strong></td>
<td>concepto: hora-llegada</td>
<td>¿A qué hora desea llegar?</td>
<td>Asks about the arrival time.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| **s_solicitud (…)** | † concepto: hora-am-pm | ¿De la mañana o de la tarde? | Asks about the AM/PM-ness of a time which we do not know. We just ask “in the morning or in the afternoon?”.
| **s_solicitud (…)** | concepto: tipo-viaje | ¿Del mismo viaje, el viaje de vuelta, u otro viaje? | Asks what kind of trip the user wants to make a query about: the same trip, the return trip, or another trip. |
5.3.4 Giving information

<table>
<thead>
<tr>
<th>s_informacion</th>
<th>(...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>concepto:</td>
<td>ciudad-origen,</td>
</tr>
<tr>
<td></td>
<td>ciudad-destino,</td>
</tr>
<tr>
<td></td>
<td>fecha-salida,</td>
</tr>
<tr>
<td></td>
<td>[fecha-llegada, hora-salida,</td>
</tr>
<tr>
<td></td>
<td>hora-llegada</td>
</tr>
<tr>
<td>ciudad-origen:</td>
<td>c₀</td>
</tr>
<tr>
<td>ciudad-destino:</td>
<td>c₁</td>
</tr>
<tr>
<td>fecha-salida:</td>
<td>d₀</td>
</tr>
<tr>
<td>[fecha-llegada:</td>
<td>d₁</td>
</tr>
<tr>
<td>hora-salida:</td>
<td>t₀</td>
</tr>
<tr>
<td>hora-llegada:</td>
<td>t₁</td>
</tr>
<tr>
<td>tipo-tren:</td>
<td>T₁₀</td>
</tr>
<tr>
<td>[servicios:</td>
<td>Tᵣ</td>
</tr>
<tr>
<td>[tiempo-recorrido:</td>
<td>+rápido</td>
</tr>
<tr>
<td>[precio:</td>
<td>+barato)</td>
</tr>
</tbody>
</table>

_Gives basic information about a single train._

<table>
<thead>
<tr>
<th>s_informacion</th>
<th>(...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>concepto:</td>
<td>estacion-origen,</td>
</tr>
<tr>
<td></td>
<td>estacion-destino</td>
</tr>
<tr>
<td>estacion-origen:</td>
<td>s₀</td>
</tr>
<tr>
<td>estacion-destino:</td>
<td>s₁</td>
</tr>
</tbody>
</table>

_Sale de place(s₀) y llega a place(s₁)._  
_Gives the departure and arrival station of a train._

<table>
<thead>
<tr>
<th>s_informacion</th>
<th>(...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>concepto:</td>
<td>tipo-tren</td>
</tr>
<tr>
<td>tipo-tren:</td>
<td>T₁₀</td>
</tr>
<tr>
<td>† tipo:</td>
<td>descripcion-est-tren</td>
</tr>
</tbody>
</table>

_Es un tren ttype(T₁₀)._  
_Gives the train type._

<table>
<thead>
<tr>
<th>s_informacion</th>
<th>(...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>concepto:</td>
<td>servicios</td>
</tr>
<tr>
<td>servicios:</td>
<td>{Tᵣᵢ}</td>
</tr>
<tr>
<td>† tipo:</td>
<td>descripcion-est-tren</td>
</tr>
</tbody>
</table>

_Cuenta con list₄(tserv(T₄ᵢ))._  
_Lists the services offered on a train._

<table>
<thead>
<tr>
<th>s_informacion</th>
<th>(...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>concepto:</td>
<td>precio</td>
</tr>
<tr>
<td>precio:</td>
<td>?</td>
</tr>
</tbody>
</table>

_El precio del billete no está disponible._  
_Says that we do not know the fare price._
5.3. FRAMES ON THE OUTPUT SIDE

<table>
<thead>
<tr>
<th>Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>s_informacion</strong> (…)</td>
</tr>
<tr>
<td><em>concepto</em>: precio, precio-iv</td>
</tr>
<tr>
<td><em>clase-billete</em>: $T_{ik}$</td>
</tr>
<tr>
<td>precio: $p$</td>
</tr>
<tr>
<td>↑ [precio-iv: $p_2$]</td>
</tr>
</tbody>
</table>

En class($T_{ik}$), el billete vale number($p$) pesetas [para sólo ida, y number($p_2$) para ida y vuelta].

Gives the fare price for a specific class.

<table>
<thead>
<tr>
<th>Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>s_informacion</strong> (…)</td>
</tr>
<tr>
<td><em>concepto</em>: precio, precio-iv</td>
</tr>
<tr>
<td>precio: $p$</td>
</tr>
<tr>
<td>↑ [precio-iv: $p_2$]</td>
</tr>
</tbody>
</table>

El billete vale number($p$) pesetas [para sólo ida, y number($p_2$) para ida y vuelta].

Gives the fare price.

<table>
<thead>
<tr>
<th>Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>s_informacion</strong> (…)</td>
</tr>
<tr>
<td><em>concepto</em>: numero-trenes</td>
</tr>
<tr>
<td>numero-trenes: 0</td>
</tr>
<tr>
<td><em>ciudad-origen</em>: $s_0$</td>
</tr>
<tr>
<td><em>ciudad-destino</em>: $s_1$</td>
</tr>
<tr>
<td><em>estacion-origen</em>: $s_0$</td>
</tr>
<tr>
<td><em>estacion-destino</em>: $s_1$</td>
</tr>
<tr>
<td><em>fecha-salida</em>: $d_0$</td>
</tr>
<tr>
<td><em>fecha-llegada</em>: $d_1$</td>
</tr>
<tr>
<td><em>intervalo.fecha-salida</em>: $d_0$</td>
</tr>
<tr>
<td><em>intervalo.fecha-llegada</em>: $d_1$</td>
</tr>
<tr>
<td><em>hora-salida</em>: $t_0$</td>
</tr>
<tr>
<td><em>hora-llegada</em>: $t_1$</td>
</tr>
<tr>
<td><em>intervalo.hora-salida</em>: $t_0$</td>
</tr>
<tr>
<td><em>intervalo.hora-llegada</em>: $t_1$</td>
</tr>
<tr>
<td><em>tipo-tren</em>: $T_{iy}$</td>
</tr>
<tr>
<td><em>servicios</em>: $T_s$</td>
</tr>
</tbody>
</table>

No hay ninguno tren [tttype($T_{iy}$)] [con tserv($T_s$)]

[de place($s_0$)] [a place($s_1$)] [que salga

[date($d_0$) $d_0$)] [time($t_0$) $t_0$] [y] [que llegue

[date($d_1$) $d_1$)] [time($t_1$) $t_1$]].

Says that no trains were found with some particular conditions.
**s_informacion**  
concepto: numero-trenes  
nnumero-trenes: n  
[calle-origen: c_0]  
[calle-destino: c_1]  
[estacion-origen: s_0]  
[estacion-destino: s_1]  
[fecha-salida: d_0]  
[fecha-llegada: d_1]  
[intervalofecha-salida:  
[ intervalo-llegada:  
[ hora-salida: t_0]  
[ hora-llegada: t_1]  
[intervalo-hora-salida: t_0]  
[intervalo-hora-llegada: t_1]  
[tiempo-recorrido: +rapido]  
[precio: +barato]  

Hay number(n) trenes [type(T_y)] con  
type(T_x) igual de rapidos [y de] baratos  
de place(c_0|c_1) a place(s_1) que sale[n]  
[date(d_0|d_1)] [time(t_0|t_1)] y [que llega[n]  
[date(d_1|d_0)] [time(t_1|t_0)].  
We must have n > 0. Says how many trains  
were found in a database query.

**s_informacion**  
concepto: hora-salida, hora-llegada, tipo-tren  
hora-salida: t_0  
hora-llegada: t_1  
nnumero-relativo-orden: i  
ttipo-tren: T_y  

El ordinal(n) es un type(T_y) sale time(t_0)  
y llega time(t_1).  
Gives basic information about a train in a list.

**s_informacion**  
concepto: tipo-tren  
tipo-tren: {T_y,i}  

Hay trena list_i(type(T_y,i)).  
Lists the available train types.

**s_informacion**  
concepto: servicios  
servicios: {T_x,i}  

Hay trenes con list_i(type(T_x,i)).  
Lists the available train services.

### 5.3.5 Explicitly requesting user confirmation

**s_confirmacion**  
tipo: explicita  
hora-salida: +pronto  
¿El que sale antes?  
Select the earliest-departing train.

**s_confirmacion**  
tipo: explicita  
hora-llegada: +pronto  
¿El que llega antes?  
Select the earliest-arriving train.
5.3. FRAMES ON THE OUTPUT SIDE

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>(…)</th>
<th>¿Un tren con tserv(T_s)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td>A train service.</td>
</tr>
<tr>
<td>servicios: T_s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>(…)</th>
<th>¿El más rápido?</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td>The fastest train.</td>
</tr>
<tr>
<td>tiempo-recorrido:</td>
<td>+rápido</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>(…)</th>
<th>¿El más barato?</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td>The cheapest train.</td>
</tr>
<tr>
<td>precio: +barato</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>(…)</th>
<th>¿Desea ir a place(v)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td>The arrival city.</td>
</tr>
<tr>
<td>ciudad-destino:</td>
<td>v</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>(…)</th>
<th>¿Desea ir a place(v)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td>The arrival station.</td>
</tr>
<tr>
<td>estacion-destino:</td>
<td>v</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>(…)</th>
<th>¿Desea salir de place(v)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td>The departure city.</td>
</tr>
<tr>
<td>ciudad-origen:</td>
<td>v</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>(…)</th>
<th>¿Desea salir de place(v)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td>The departure station.</td>
</tr>
<tr>
<td>estacion-origen:</td>
<td>v</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>(…)</th>
<th>¿Desea salir date(v)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td>The departure date.</td>
</tr>
<tr>
<td>fecha-Salida:</td>
<td>v</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>(…)</th>
<th>¿Desea salir date(v)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td>The departure date interval.</td>
</tr>
<tr>
<td>intervalo-fecha-salida:</td>
<td>v</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>(…)</th>
<th>¿Desea salir date(v)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td>The departure day type.</td>
</tr>
<tr>
<td>tipo-dia-salida:</td>
<td>v</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>(…)</th>
<th>¿Desea llegar date(v)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td>The arrival date.</td>
</tr>
<tr>
<td>fecha-llegada:</td>
<td>v</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>(…)</th>
<th>¿Desea llegar date(v)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td>The arrival date interval.</td>
</tr>
<tr>
<td>intervalo-fecha-llegada:</td>
<td>v</td>
<td></td>
</tr>
</tbody>
</table>
### 5.3.6 Implicitly requesting user confirmation

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>¿Desea llegar date(v)?</th>
<th>The arrival day type.</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hora-llegada:  v</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>¿Desea salir time(v)?</th>
<th>The departure time.</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hora-salida:   v</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>¿Desea salir time(v)?</th>
<th>The departure time interval.</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td></td>
</tr>
<tr>
<td>intervalo-hora-salida:  v</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>¿Desea llegar time(v)?</th>
<th>The arrival time.</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hora-llegada:  v</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>¿Desea llegar time(v)?</th>
<th>The arrival time interval.</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td></td>
</tr>
<tr>
<td>intervalo-hora-llegada:  v</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>¿El ordinal(v)?</th>
<th>Select a train by its position in a list, or scroll up and down.</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td></td>
</tr>
<tr>
<td>numero-relativo-orden:  v</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>¿Ha dicho de am-pm-name(v)?</th>
<th>An AM/PM word.</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hora-am-pm:    v</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>¿Un ttype(v)?</th>
<th>A train type.</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: explícita</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tipo-tren:     v</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>El que sale antes.</th>
<th>The earliest-leaving train.</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: implicita</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hora-salida:   +pronto</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>El que llega antes.</th>
<th>The earliest-arriving train.</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: implicita</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hora-llegada: +pronto</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_confirmacion</th>
<th>Un tren con tserv(T),</th>
<th>A train service.</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo: implicita</td>
<td></td>
<td></td>
</tr>
<tr>
<td>servicios:     T,</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.3. FRAMES ON THE OUTPUT SIDE

<table>
<thead>
<tr>
<th>Frame</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s-confirmacion</td>
<td>El más rápido. Select the fastest train.</td>
</tr>
<tr>
<td>tipo: implícita</td>
<td>tiempo-recorrido: +rápido</td>
</tr>
<tr>
<td>s-confirmacion</td>
<td>El más barato. Select the cheapest train.</td>
</tr>
<tr>
<td>tipo: implícita</td>
<td>precio: +barato</td>
</tr>
<tr>
<td>s-confirmacion</td>
<td>A place(v). The arrival city.</td>
</tr>
<tr>
<td>tipo: implícita</td>
<td>ciudad-destino: v</td>
</tr>
<tr>
<td>s-confirmacion</td>
<td>A place(v). The arrival station.</td>
</tr>
<tr>
<td>tipo: implícita</td>
<td>estacion-destino: v</td>
</tr>
<tr>
<td>s-confirmacion</td>
<td>De place(v). The departure city.</td>
</tr>
<tr>
<td>tipo: implícita</td>
<td>ciudad-origen: v</td>
</tr>
<tr>
<td>s-confirmacion</td>
<td>De place(v). The departure station.</td>
</tr>
<tr>
<td>tipo: implícita</td>
<td>estacion-origen: v</td>
</tr>
<tr>
<td>s-confirmacion</td>
<td>Salir date(v). The departure date.</td>
</tr>
<tr>
<td>tipo: implícita</td>
<td>fecha-salida: v</td>
</tr>
<tr>
<td>s-confirmacion</td>
<td>Salir date(v). The departure date interval.</td>
</tr>
<tr>
<td>tipo: implícita</td>
<td>intervalo-fecha-Salida: v</td>
</tr>
<tr>
<td>s-confirmacion</td>
<td>Salir date(v). The departure day type.</td>
</tr>
<tr>
<td>tipo: implícita</td>
<td>tipo-dia-salida: v</td>
</tr>
<tr>
<td>s-confirmacion</td>
<td>Llegar date(v). The arrival date.</td>
</tr>
<tr>
<td>tipo: implícita</td>
<td>fecha-llegada: v</td>
</tr>
<tr>
<td>s-confirmacion</td>
<td>Llegar date(v). The arrival date interval.</td>
</tr>
<tr>
<td>tipo: implícita</td>
<td>intervalo-fecha-llegada: v</td>
</tr>
<tr>
<td>s-confirmacion</td>
<td>Llegar date(v). The arrival day type.</td>
</tr>
<tr>
<td>tipo: implícita</td>
<td>tipo-dia-llegada: v</td>
</tr>
</tbody>
</table>
### Description of frame types, case names and case values

In this section we list and describe all the frame types, case names, and case values for the Basurde[lite] frames. Some of this is already defined in [31, 44], but we had to make some additions and changes to accommodate the Basurde[lite] dialogues.

The case values for date and time expressions are described in a separate sections due to their complexity (sections 5.4.4, 5.4.5, 5.4.6, 5.4.7).

#### Accents and capital letters

The modules have been designed to deal with both capitalized and non-capitalized words and, in most cases, both with accented and non-accented words. However, we decided to follow these conventions:

1. All words are in all-lowercase.
2. Frame types and case types, and case values which have a special meaning, are written without accents or special letters. Accents are removed in the trivial way. Other letters get changed as follows: “ñ” → “n”, “Ñ” → “N”, “ç” → “c”, “Ç” → “C”, “.” → “.”.

3. The remaining case values, that is, names of places, train services, train types, and parts of the day, keep their accents and special letters.

In case of doubt, the authoritative sources are the application definition files (appendix C on page 207).

5.4.2 Frame types

u_fallo-reconocimiento Signals a problem in the speech recognizer, probably a recognition failure, or other kinds of system failures. While this is not anything that the user says, it is a direct consequence of a user utterance, and treating this as a frame permits a more smooth module integration.

u_no-entendido The user had trouble understanding what the system said.

s_no-entendido The system couldn’t understand what the user said.

u_informacion The user gives some information to the system, or makes an assertion.

s_informacion The system gives some information to the system.

s_solicitud The system requests some information from the user.

s-confirmacion The system gives the user some information that it believes to be correct, expecting the user to tell the system if the information was right or wrong; that is, this is a request for confirmation. There are two kinds of confirmation requests: In explicit requests, the system explicitly asks the user whether the information is right or wrong; the user should answer that, and maybe also correct it. In implicit requests, the system simply asserts the information; if the user doesn’t complain, the system assumes it to be correct.

s_cortesia The system makes some offer to the user.

5.4.3 Case names and values

We give a description of each case name, and the list of its valid values.
ciudad-origen  Gives the departure city. Per [31], different cases are used for cities and for train stations. The list of valid city values can be found in the file \texttt{comprensio/places.txt} (see C.3.17 on page 394); they were defined in [4, 3]. The way to obtain the user utterances associated with these is described in \texttt{gram_recon/processa-bnf.sh} (see C.1.6 on page 234).

ciudad-destino  Gives the destination city.

estacion-origen  Gives the departure train station. Per [31], different cases are used for cities and for train stations. The list of valid station values can be found in the file \texttt{comprensio/places.txt} (see C.3.17 on page 394); they were defined in [4, 3].

estacion-destino  Gives the destination train station.

fecha-salida  Gives the departure date, when this is a single date. Values for dates follow a complex scheme which is an extension of the Basurde standard †. It is described in section 5.4.4 (page 106).

A special value of “?” † is used when the information is not the date itself, but that we are talking about a departure date.

fecha-llegada  Gives the arrival date, when this is a single date. The same values as for \textit{fecha-salida} are accepted.

intervalo-fecha-salida  Gives the departure date, when this is date interval. Values for date intervals are also an extension of the Basurde standard described in section 5.4.5 (page 108) †.

interval-fecha-llegada  Gives the arrival date, when this is a date interval. The same values as for \textit{intervalo-fecha-salida} are accepted.

tipo-dia-salida  Gives the departure date as a kind of day. Valid values are \textit{festivo} for holidays and \textit{laborable} for workdays.

tipo-dia-llegada  Gives the arrival date as a kind of day. The same values as for \textit{tipo-dia-salida} are accepted.

hora-salida  Gives the departure time, when this is a single time. Values for times follow a complex scheme which is an extension of the Basurde standard †. It is described in section 5.4.6 (page 110).

A special value of “?” † is used when the information is not the time itself, but that we are talking about a departure time. A special value of \textit{pronto} means “the earliest” or “soonest” († this value is defined in Basurde, but has a different meaning).
hora-llegada Gives the arrival time for single times. Uses the same values as hora-salida.

intervalo-hora-salida Gives the departure time for time intervals. Values also follow an extension to the Basurde standard †, and are described in section 5.4.7 (page 111).

A special value of “?” † is used when the information is not the time itself, but that we are talking about a departure time.

intervalo-hora-llegada Gives the arrival time for time intervals. Uses the same values as intervalo-hora-salida.

hora-am-pm† Gives an expression for a part of the day, used in determining the AM/PM-ness of times and other time information. Valid values are madrugada† (dawn), mañana† (morning), mediodía† (noon), tarde† (afternoon), noche† (night), and also am† and pm† for a general form of AM and PM.

tipo (for s_cortesia frames) Gives the kind of politeness expression. Use cierre for a closing expression ("goodbye"), entrada for a greeting, and espera to ask the user for some patience. Use en-repeticion† to signal that the following frames repeat previously given information.

The oferta “tipo” is used for a larger class of offerings. For these, a concepto† case indicates the precise offering being made:

+rapido† The system offers to search for the fastest train.

instrucciones† The system offers a short tutorial to the user (who must decide if she wants to hear it or not).

mas-info-en-lista† The system offers to look for more information about one of the trains shown in the previous list.

mas-info-este-tren† The system offers to give more information about this train.

nueva-consulta† The system asks the user whether she wants to make another query.

numero-relativo-orden† The system lets the user select the previous or the next train in a list.

The instrucciones† “tipo” is used to actually give the user (not just offer) the short tutorial. Probably, this should not be in a s_cortesia frame, but we did not find any better place.

tipo (for s_confirmacion frames) Gives the kind of confirmation request. Use explicita for explicit confirmation requests, and implicita for implicit confirmation requests. No other values are allowed†.
tipo (for u_no-entendido frames)† Gives the kind of problem that the user is having. A transferir-cierre† means that the user requests the call to be transferred to a human operator. A repeticion† means that the user wants the system to repeat its last interesting utterance. A timeout† is sent when the speech recognizer detects a user timeout. A palabra-clave-error† is used when the user wants to bring the dialogue back to a known state (this "known state" is actually the beginning of the dialogue).

tipo (for s_no-entendido frames) Gives the kind of problem that the system is having. A cierre† means that the system will finish the dialogue and transfer the call to a human operator, because there were too many errors. A repeticion is a system request for the user to repeat what she just said. A error-sistema† means that there was an internal error in the Basurde[lite] system, and that this call will be transferred to an operator. A falta-informacion† is given by the system when the user refuses to answer important questions.

tipo (for u_fallo-reconocimiento frames)† Gives the degree of recognition failure. Use duda† for a partial recognition failure and total† for a total recognition failure.

tipo (for s_informacion frames with concepto=servicios or concepto=tipo-tren)† If this is given and set to descripcion-este-tren, the system’s utterance will be modified to reflect the fact that the list of services being given does not belong to a collection of trains but to a specific train.

clasa-billete Gives the ticket class (first class, economy, ... ). The used values can be found in dialog/cgi-class.txt (see C.4.31 on page 646). They incorporate some of the values defined in [4, 31], but not all †.

precio Gives the price for a one-way ticket. Valid values are a (possibly decimal) non-negative number, or the word barato which is used in requests for the cheapest train. The special value “?” † is used to indicate that the price is not available.

precio-iv† Gives the price for a return ticket. Only non-negative numbers are allowed as values.

servicios Gives one or more train services. If more than one service is given, they should be separated by commas (and no space). The used values can be found in dialog/cgi-serv.txt (see C.4.33 on page 651). They are defined in [4, 31]. The value cômodo, defined in Basurde, is not used for Basurde[lite] †. The value cualquiera† means the user doesn’t know which service she prefers.
5.4. FRAME TYPES, CASE NAMES AND CASE VALUES

**tipo-tren** Gives one or more train types. If more than one is given, they should be separated by commas (and no space). The used values can be found in `dialog/cgi-traintype.txt` (see C.4.34 on page 651). The value *cualquiera*† means the user doesn’t know which train type she prefers.

We use the ones defined in [4], not those defined in [31] †. The former have many different forms for the same type of train, depending on how the user said them; the latter has only one form for each type of train. Using [4] we can make the dialogue more natural by replying to the user in the same way that she talks to us.

**tiempo-recorrido** Gives the travel time. The only valid value is *rapido* meaning the train with the shortest travel time. Numerical values are not allowed †.

**numero-relativo-orden** Selects a train by its position in a list. Valid values are *primero* (first), *segundo* (second), *tercero* (third), *ultimo* (last), *anterior* (previous) and *siguiente* (next). Some Basurde values are not used (*cuarto* and *quinto*) †.

**numero-trenes** Gives how many trains were found in a database query. Valid values are integer non-negative numbers.

**criterio-reduccion**† Gives the criterion that the user wants to use in selecting a specific train in a long list. Valid values are *hora*† for selecting trains based on their timetable, *servicios*† for their services, and *tipo-tren*† for their types.

**rango-ordinales**† For an *s-cortesia* frame with *tipo-*oferta and *concepto-*numero-relativo-orden, this selects which ordinal range the question should be about. These only change the way the question is asked, but not which answers are accepted by the system.

**tipo-viaje**† Indicates which kind of trip the user is interested in getting more information about. The following values are accepted: *mismo*† for the same trip, *vuelta*† for the return trip, and *otro*† for another trip.

**si-no**† Gives the answer to a yes/no question or a confirmation request. The value *si*† means “yes”, and the value *no*† means “no”.

**concepto** (for *s-solicitud frames*)† Tells which information the system is actually asking. Other frames may be given to complement the question, but they are not asked about. More than one item may be given as value; in this case, they must be separated by commas (without spaces). Valid
value elements are: tipo-tren, servicios, hora-am-pm, ciudad-destino, ciudad-origen, hora-salida, hora-llegada, fecha-salida, fecha-llegada, tipo-viaje\textsuperscript{†}, numero-relativo-orden\textsuperscript{†} and criterio-reduccion\textsuperscript{†}.

concepto (for s\textsubscript{information} frames)\textsuperscript{†} Tells which information the output sentence should be focused on. Additional information may be given by other frames, but the sentence is not about it. More than one item may be given as value; in this case, they must be separated by commas (without spaces). Valid value elements are: ciudad-destino, ciudad-origen, estacion-destino, estacion-origen, hora-salida, hora-llegada, fecha-salida, fecha-llegada, servicios, precio, precio-iv\textsuperscript{†}, and numero-trenes.

To illustrate what difference it makes for a case name to be listed in the concepto case, consider a s\textsubscript{information} frame with cases ciudad-origen: Barcelona and hora-salida: 13.00. If there is a concepto: ciudad-origen, hora-salida case, the resulting text (in English) would look like “The train leaves from Barcelona at 13.00”. On the other hand, if the concepto case were just concepto: hora-salida, the resulting text would be “The train from Barcelona leaves at 13.00”.

5.4.4 Case values for single dates

Values in date cases for single dates in Basurde\textsuperscript{lite} not only contain the date itself, but also additional information. For input frames, a description of how the user gave this date, and which parts were explicitly given by the user and which were inferred by the system. For output frames, a description on how the date is to be converted to a natural language expression.

Knowing how the user gave a date allows the system to respond to the user in her same style. This makes the dialogue more natural and avoids surprising the user.

The Basurde standard specifies [31] that date expressions have the format

\texttt{DD-MM-AAAA}

where \texttt{DD} is a 2-digit value for the day, \texttt{MM} is a 2-digit value for the month, and \texttt{AAAA} is a 4-digit value for the year.

In Basurde\textsuperscript{lite}, date expressions on the input side (what is recognized from the user) have the format

\texttt{DD-MM-AAAA [Style] [WhatGiven]}

and those on the output side

\texttt{DD-MM-AAAA [Style]}

where \texttt{DD}, \texttt{MM} and \texttt{AAAA} give the day, month and year, but need not have exactly 2 digits (or 4 for the year); \texttt{Style} describes the style that the user followed in giving the date, or the style that the system should use in saying the date; and
5.4. FRAME TYPES, CASE NAMES AND CASE VALUES

WhatGiven tells exactly which parts of the date were given by the user. The frames produced by the system do have exactly 2 digits for DD and MM, and 4 for YYYY.

In some cases, described above, “?” is also a valid date value for input frames.

The following format specifiers are accepted as Style parameters, both for input and output frames:

D Day only. El día tres.

DM Day and month (as January, February, … ). El tres de enero.

Dm Day and month (as number). El tres del uno.

+0 Today. Hoy.

+1 Tomorrow. Mañana.

+2 The day after tomorrow. Pasado mañana.

S Day of the week only (Monday, Tuesday, … ).

SD Day of the week and day of the month.

SDM Day of the week, day of the month, and month as word.

p:x Where x is “D”, “DM”, “Dm”, “S”, “SD”, “SDM”, says “next” followed by the date using x as style. For example, “p:S” could produce “next Monday”.

The following format specifiers are accepted as Style parameters, for input frames only:

X The user has given a full date without explicitly giving a day of the month or a month (for example, Christmas). The system will respond with a “D” or “DM” format.

d+1 or d-1 The user said “the next day” or “the previous day”. The system is able to fill in the day, month and year fields because there is an internal “reference” date.

For compatibility with Basurde modules, the ORG module also accepts Basurde date values. In this case, a style “DM” is assumed.

For “+0”, “+1” and “+2” formats, the ORG module doesn’t really check that the given date is indeed today, tomorrow, or the day after tomorrow.
The *WhatGiven* field, found in input frames only, tells exactly which kind of information about the date was really given by the user. It is a string of one or more specifiers, concatenated together in any order.

For example, if the user says “next Tuesday”, the system will give the full date for next Tuesday, but will signal that the user only gave the day of the week with a “S” *WhatGiven* field, for example, 27–02–2001 [S] [S] .

The following specifiers are accepted:

**X** Full date, not given by means of a day number and an optional month. For example, *Christmas*.

**D** Day number.

**M** Month (as a number or a word).

**S** Day of the week.

**c** The user didn’t give a specific date, but a date class. For example, with *any Friday* the system will behave as if she had said *next Friday*, but will signal the difference with this “c”.

The *WhatGiven* field may also be left blank. This means that it was not possible, for some reason, to determine how much information the user gave.

### 5.4.5 Case values for date intervals

Values in date cases for date intervals also contain additional information. For input frames, a description of how the user gave this date, and which parts were explicitly given by the user and which were inferred by the system. For output frames, a description on how the date is to be converted to a natural language expression.

The Basurde standard specifies [31] that date interval expressions have the format

```
DD–MM–AAAA/DD–MM–AAAA
```

where *DD* is a 2-digit value for the day, *MM* is a 2-digit value for the month, and *AAAA* is a 4-digit value for the year. The interval starts on the first date given and ends on the second date.

In Basurde[lite], date interval expressions on the input side (what is recognized from the user) have the format

```
DD–MM–AAAA/DD–MM–AAAA [Style] [WhatGiven]
```

and those on the output side

```
DD–MM–AAAA/DD–MM–AAAA [Style]
```
5.4. FRAME TYPES, CASE NAMES AND CASE VALUES

where fields have the same meaning as for single dates. As before, the modules can handle DD, MM and AAAA fields with not exactly 2 (4 for the year) digits.

The following format specifiers are accepted as Style parameters, both for input and output frames:

a-b Where a and b are valid style fields for single dates: the date interval should be said (or has been said) using its starting date with style a and its ending date with style b.

p:fs “Next weekend”. El próximo fin de semana. For output frames, the system does not really check that the dates correspond to the next weekend.

fs:a With a date interval for a weekend (which, of course, contains a Saturday), says “the weekend of x”, where x is that one Saturday. The date for Saturday is given with style a, which must be “D” or “DM”. El fin de semana del doce.

The following format specifiers are accepted as Style parameters, for input frames only:

fs The user said “the weekend”. This is not the same as “next weekend”, though the system may process them in the same way.

For compatibility with Basurde modules, the ORG module also accepts Basurde date interval values. In this case, a style “DM-DM” is assumed.

The WhatGiven field, found in input frames only, should tell exactly which kind of information about the date was really given by the user. It is a string of one or more specifiers, concatenated together in any order.

Currently, this field is not used anywhere.

The following specifiers are accepted:

X This is supposed to mean “full date interval, not given by means of day numbers and optional months”, but, currently, all date intervals get this specifier.

c The user didn’t give a specific date interval, but a date interval class. For example, with any weekend the system will behave as if she had said next weekend, but will signal the difference with this “c”.

The WhatGiven field may also be left blank. This means that it was not possible, for some reason, to determine how much information the user gave.
5.4.6 Case values for single times

Values in time cases for single times in Basurde[lite] not only contain the time itself, but information about the format for that time in natural speech. For input frames, this is how the user gave the time; for output frames, this is the format in which the time should be given.

The Basurde standard specifies [31] that time expressions have the format

HH.MM

where HH is a 2-digit value for the hour and MM is a 2-digit value for the minute. In Basurde[lite], single time expressions (both input and output) have the format

HH.MM [Style]

where HH and MM give the hour and minute, but need not have exactly 2 digits, and Style describes the style that the user followed in giving the time, or the style that the system should use in saying the time. The frames produced by the system do have exactly 2 digits for HH and MM.

In some cases, described above, “?” is also a valid time value for input frames.

The following format specifiers are accepted as Style parameters, both for input and output frames:

trad!m Traditional format, without information about AM/PM-ness. The hour field should be taken modulo 12. That is 13:15 [trad!m] and 01:15 [trad!m] both give the same time, “quarter past one” (la una y cuarto).

trad Traditional format, with information about AM/PM-ness. In this case, 13:15 and 01:15 have different expressions.

24 Numerical format, with 24 hour time. The hour value has information about AM/PM-ness. For example, 14:15 [24] gives “fourteen fifteen” (las catorce quince).

12 Numerical format, with 12 hour time and explicit AM/PM-ness. The hour value has information about AM/PM-ness. For example, 14:15 [12] gives “two fifteen PM” (las dos quince de la tarde).

12!m Numerical format, without AM/PM-ness information in the hour value. The hour value, therefore, should be taken module 12. For example, 14:15 [12!m] give “two fifteen” (las dos quince). AM/PM-ness is not given because it is not known at that moment.

The following format specifiers are accepted as Style parameters, for input frames only:
The user gave a full time without giving an explicit hour value and an
optional minute value. For example, she said at noon. The system
will choose the appropriate format to respond to these frames, proba-
bly “trad” or “trad!m”. The hour value in this frame has the correct
AM/PM-ness.

For compatibility with Basurde modules, the ORG module also accepts Basurde
single time values. In this case, a style “trad” is assumed.

5.4.7 Case values for time intervals

Values in time cases for time intervals in Basurde[lite] not only contain the
starting and ending times, but information about the format for that time
interval in natural speech. For input frames, this is how the user gave the time;
for output frames, this is the format in which the time should be given.

The Basurde standard specifies [31] that time interval expressions have the
format

\[ HH, MM\rightarrow HH, MM \]

where \( HH \) is a 2-digit value for the hour and \( MM \) is a 2-digit value for the minute;
the starting time is the time given first, and the ending time is the second one.

In Basurde[lite], time interval expressions (both input and output) have the
format

\[ HH, MM\rightarrow HH, MM \ [Style] \]

where \( HH \) and \( MM \) give the hour and minute, but need not have exactly 2 digits,
and \( Style \) describes the style that the user followed in giving the time, or the
style that the system should use in saying the time. The frames produced by
the system do have exactly 2 digits for \( HH \) and \( MM \).

In some cases, described above, “?” is also a valid time value for input frames.

The following format specifiers are accepted as \( Style \) parameters, both for input
and output frames:

\( a-b \) (where \( a \) and \( b \) are valid style fields for single times). The time interval
should be said (or has been said) using its starting time with style \( a \) and
its ending time with style \( b \).

\( <=:b \) (where \( b \) is a valid style field for a single time). This is a time interval
with the form “before … “, using the ending single time, with style \( b \), as
the limit. For example, 12.00–14.00 \([<=:\text{trad!m}]\) gives “before two
o’clock” (antes de las dos en punto).

\( >=:a \) (where \( a \) is a valid style field for a single time). This is a time interval
with the form “after … “, using the starting single time, with style \( a \),
as the limit. For example, 10.00–14.00 [>=:trad!m] gives “after ten o’clock” (después de las diez en punto).

\(\sim=:x\) (where \(x\) is a valid style field for a single time). This is a time interval with the form “around \(\ldots\)”, using the average of the starting and ending single times, with style \(x\), as the central time. For example, 10.05–14.05 ["=::24] gives “around twelve oh five” (hacia las doce y cinco).

The following format specifiers are accepted as Style parameters, for input frames only:

\[\text{X}\] The user gave a full time interval without giving explicit hour values and optional minute values. For example, she said in the morning. Some of these expressions are normalized in [44], and some are extensions to the Basurde standard. The system will choose the appropriate format to respond to these frames, probably “trad-trad” or “tradm-tradm”. The hour values in this frame have the correct AM/PM-ness.

For compatibility with Basurde modules, the ORG module also accepts Basurde time interval values. In this case, a style “trad-trad” is assumed.

5.4.8 Recognition states for the NLU module

The dialogue manager sends control packets to the NLU module which tell it what kind of sentences it should expect and try to analyze, and also give it information which it needs to analyze some sentences (for example, the default date to use if the user gives a relative date expression).

We describe here the possible NLU states and the associated state variables, for the Basurde[lite] application.

State variables

The following recognition states are defined:

**explicit** The user replies to an explicit confirmation request. This state must be combined with another state (except implicit), which indicates what the user is responding to.

**implicit** The user replies to an implicit confirmation request. This state must be combined with another state (except explicit), which indicates what the user is responding to, in case the implicit confirmation is accepted.

**am_pm** The user should clarify whether a system-given time is an AM time or PM time.
place The user should give a city or station name.

date The user should give a date, date interval, type of day, or any other
date-related information.

time The user should give a time, time interval, or any other time-related
information.

train_type The user should give a type of train.

train_services The user should give a service which may be available on trains.

crit_red The user is choosing a reduction criterion.

sel_list The user is directly selecting a train in a short list.

viatge The user is choosing a kind of trip.

In some recognition states, the NLU module needs to know some extra informa-
tion:

tofrom For place, date, and time states. This should be to or from. If the
user reply does not indicate whether she means the departure or arrival
place, date or time, this value is used instead (to for arrival, from for
departure).

refdate For date states, the date that must be used as “now” if the user gives
a date in relative form (such as three days after). It must be a Tcl 3-
tuple containing a year (4-digit), a month (in numeric form) and a day,
for example {2001 4 25}.

isitime For time states, a 1 value indicates that the answer should be a time
interval, and a 0 means it should be a point in time. This is only used
if the user does not give enough information to distinguish between both
cases.

5.5 Dialogue manager script files

5.5.1 Node class hierarchy

As we describe in (4.3.2 on page 68), the dialogue manager lets us define node
classes to implement the Basurde[lite] dialogue strategy. We needed to define
several of them. These are shown in figures 5.2 to 5.5.

The Application class (figure 5.2) is the main class for the Basurde[lite] dialogue
application: The node tree is an instance of this class. It is a container node
Figure 5.2: Node classes: the Application class
which holds nodes and node subtrees for all the user-given information for a certain transaction: the departure and arrival place (\textit{Place}), the travel date (\textit{HLDate}), and optionally the time (\textit{HLTime}), type, and other restrictions given by the user when the result set for the desired date is too large.

The subtrees for the departure and arrival places and for the travel date are fixed children of an Application node. Children of the remaining classes are added dynamically as the dialogue requires it —for example, a TrainType child will only be added if the system must really have the user decide a train type.

In the same figure we also show the Boolean, MultipleChoice, ListSelection and ReductionCrit classes.

A Boolean node is one about which a yes/no question is asked (for example, \textit{Do you want more information?}), without using the confirmation mechanism: the first recognized answer is final. The developer indicates which question should be asked for each Boolean node.

For MultipleChoice nodes, a multiple-choice question is asked (for example, \textit{What do you prefer, a fast train, a cheap train, or an overnight train?}). The confirmation mechanism is also not used. The developer indicates which question should be asked, and which frame patterns should be accepted for each possible choice.

There are two specializations of the MultipleChoice class (conceptually, they do not have corresponding \texttt{constructor} procedures in the Tcl sources): a ListSelection node class, for letting the user choose a train in a short list, and a ReductionCrit, for letting the user choose a result set reduction criterion.

The TrainType node class is used to tell the user which train types were found, and let her choose one. The confirmation mechanism is used here. The list of available train types is given as a node variable, \texttt{available}.

Similarly, the TrainService node class is used to tell the user which train services (sleeping car, restaurant, TV, ...) are available, and let her choose one.

A Place node (figure 5.3) is used to ask the user about a place (departure or arrival place).

This place can be given as a train station or as a city. Two children nodes, of types Station and City respectively, handle these two cases. Initially, the \texttt{city} node is active, so the user will be asked about a city. However, she may reply with a station. in this case, the \texttt{station} node will be activated, and input frame will be rerouted to it. The parent node (Place) handles this rerouting, and maintains, as node variables (\texttt{case} and \texttt{case\_as\_city}) the representation of the current result as a case-value pair (as the user gave it, for \texttt{case}, and converted into a city, for \texttt{case\_as\_city}).

All these nodes store in a node variable whether they should ask about the departure or the arrival place, since different questions need to be asked for
each case. This information is given to them by their parent node when they are instantiated.

A HLDDate node (figure 5.4) is used to ask the user about the departure or arrival date. It has two children, instances of Date, one for the departure date and one the arrival date. Initially, the departure date is active, so that is what the user will be asked. However, she may reply with an arrival date, in which case the departure date tree will be deactivated, the arrival date tree will be activated, and the input frame will be rerouted to the arrival date tree.

The HLDDate node keeps, in the case node variable, a representation of the user’s reply as a case-value pair. Another node variable, prefers_arrival_date,
5.5. **DIALOGUE MANAGER SCRIPT FILES**

...tells whether the user gave a departure or arrival date as her last answer for this subtree. This will be used, if the system ever has to ask a date again (for instance, if the user requests information about another trip), to know if we should ask first about the departure or the arrival date. Of course the difference only makes sense for overnight trips.

Each Date node has three children, instances of DateSingle, DateInterval and DayType. These correspond to the three different date kinds: single date, date interval, and day type respectively. The same full question is asked for all of them, but confirmation questions differ. User replies are routed to the correct one. The user may reply with a different kind of date than the one she was asked about.

These Date nodes and their children store in a node variable whether they should ask about the departure or the arrival date, since different questions need to be asked for each case. This information is given to them by their parent HLDDate node when they are instantiated.

---

![Diagram](image.png)

**Figure 5.5: Node classes: the Time class**

To ask about departure or arrival times, the HLTTime node class should be used (figure 5.5). It is very similar in structure and general behavior to the HLDDate class. It has two children, instances of Time, one for the departure time and one the arrival time. Initially, the departure time is active, so that is what the user will be asked. However, she may reply with an arrival time.

The HLTTime node keeps, in the case node variable, a representation of the user's reply as a case-value pair. The node does not remember (at least not in the way that HLDDate nodes do) whether the user gave a departure or an arrival time; we thought that, ergonomically, there was no need to do so.

Each Time node has two children, instances of TimeSingle and TimeInterval. These correspond to the two different time kinds: single time and time interval respectively. The same full question is asked for all of them, but confirmation questions differ. User replies are routed to the correct one. The user may reply...
with a different kind of time than the one she was asked about.
Each Time node keeps in the node variable \textit{train\_times} the departure or arrival times (depending on the node) of all trains returned by the database query. This is used to resolve AM/PM-nos ambiguity. For example, if there is a train at 03:00 and another at 15:00 and the user says “3 o’clock” as a time, she will be asked to choose between “3 in the morning” and “3 in the afternoon”. If the 15:00 train weren’t there, the system will assume 03:00 and ask the user to confirm it.

These Time nodes and their children store in a node variable whether they should ask about the departure or the arrival time, since different questions need to be asked for each case. This information is given to them by their parent HLTime node when they are instantiated.

\subsection{Frame acceptors}

In figures 5.6 to 5.9 we show which frames are accepted at each node in a Basurde[lite] application. We show accepted frames in \textit{italics} below each node. For \textit{u\_informacion} frames, the frame type is not given.

Some frames are accepted at the C++ level, by the dialogue manager, instead of the Tcl level. The dialogue manager is shown as just one more node, even if it is not one.

Frames in red are accepted by \textit{multiplexor nodes}: they actually belong to one child, but can be received from any child; if they are received by the wrong child, this one is disabled and the right child is activated. Then the frame is rerouted to the correct child.

Figure 5.6 gives the accepted frames for the dialogue manager, the root Application node, and TrainType, TrainService, Boolean, ListSelection and ReductionCrit nodes.

Figure 5.7 gives the accepted frames for an HLDate tree.

Figure 5.8 gives the accepted frames for an HLTime tree.

Figure 5.9 gives the accepted frames for a Place tree.

The frames accepted by a MultipleChoice node depend on the arguments given in its instantiation. Abstract MultipleChoice nodes are not shown here.

\section{Multiple languages}

Not just Spanish is spoken in Spain. Other languages that are relevant to this application are Catalan, Galician, and Basque. Also, some stations and cities
Figure 5.6: Frame acceptors: the top-level
Figure 5.7: Frame acceptors: dates

Figure 5.8: Frame acceptors: times
accepted in the Basurde[lite] application are outside Spain.

We thought that it would be interesting if the whole system could work with multiple-language versions of the names of these places. Even if the system uses Spanish for its main expressions, it is usually considered a sign of good will to use the same language as the user does for, at least, place names.

We could simply have given more than one transcription for each of these names. With this, the speech recognizer would be able to understand more than one version of the place. However, we would not get information about the language that the user used, and we would not know how to talk back to the user. Instead, what we did was create new Basurde place identifiers for these places. We followed this convention:

For each place with a Basurde identifier \( x \) where the Spanish name is not the only relevant name, we make a new Basurde identifier \( \_\text{lang} \_nn\_x \), where \( nn \) is the two-letter language identifier for the relevant language [25] (and see also table 5.1). Case is not significant. The identifier without this prefix corresponds to the Spanish translation of the name of the place. The identifier with this prefix corresponds to the name in the original language.

The speech recognizer available for Basurde[lite] is capable of recognizing Spanish and Catalan. For the remaining languages, we gave an approximate transcription of the name in the original language using Spanish spelling.

The speech synthesizer is, theoretically, also capable of synthesizing both Spanish and Catalan. However, we found out that there were some audible glitches
at each language change, something which would be acceptable in an inter-
sentence pause but not within a sentence. Therefore we decided to use the
synthesizer only in Spanish mode, and give approximate spellings in Spanish
for all the other languages.

Since the place identifiers required by the database query module are different
from the standard Basurde place identifiers, we do a translation at the dialogue
manager module. It is therefore trivial to extend this translation to the new
identifiers. However, as far as the rest of the system is concerned, the two
versions of a place identifier are two different places.

For some obscure reason regarding the speech recognizer input format which
is further explained in C.1.1 (page 207), some words actually carry a language
prefix for Spanish (es).
Part II

Results
Chapter 6

Evaluation

There were two goals for this project:

1. To be able to complete a dialogue system for accessing the same information that Basurde does in much less time than it will take to develop Basurde, and

2. by using a smaller and simpler language, to get higher recognition and language understanding success rates than for the Basurde system, and therefore to get a higher overall dialogue success rate.

The first goal has been clearly achieved. The total development time has been of around 6 months. This includes the design and implementation of the three main modules of Basurde[lite], the evaluation process, and also the adaptation of the audio, speech recognition, and speech synthesis modules from Basurde to work in the Basurde[lite] setting. In contrast, the development of the Basurde project has been going on for far more than six months and is still not complete. Since the Basurde project has not yet been completed, we cannot compare the dialogue success rates of Basurde[lite] and Basurde. However, some tests were carried out to, at least, obtain some performance measures of Basurde[lite].

The test setup consists of a full Basurde[lite] system attached to a public ISDN line. This can be accessed from any public telephone. The communications control loops were modified so that interesting transmissions between modules were logged. A total of 30 calls were made to the system by 10 testers, 3 calls per experimenter. 7 testers had experience in speech technologies, but not in Basurde[lite]. The author was one of the testers, and made 3 calls. The calls were recorded as audio.

In these calls, the subjects had to find out specific train timetable information according to some scenarios that were presented to them. The scenarios tried to reflect real situations.
One of the testers had a non-cooperative behavior. We have not included those tests in the results.

6.1 Description of measured quality parameters

From the analysis of these calls, the following parameters about individual modules were calculated:

1. Speech recognizer turn success rate: The ratio of dialogue turns in which the speech recognizer correctly recognizes the user utterance, to the total number of dialogue turns where this module operates. Since the recognition grammar allows the user to insert arbitrary speech in certain positions (thanks to the wild-card non-terminal), "correct recognition" does not mean that the recognized sentence is exactly equal to the user utterance; instead of using equality at the character level, we use equivalence at the semantic level.

For example, if the user says "I would like to go to Los Angeles", a correct recognition would be "@f go to @f los angeles", assuming a simple enough grammar.

2. Speech recognizer word success rate: The ratio of recognized words that correctly correspond to the user utterance, to the total number of recognized words excluding silence markers (SIL). Since the recognition grammar is made up of whole sentences only, this measure is not as meaningful as in other systems, for example those using n-grams. Also, we count as "word" what the speech recognizer outputs. Given the way we built the grammars, some "words" may correspond to a phrase. All this may however be useful for comparison to other systems.

For example, if the user says "I would like to go to Los Angeles", then the recognition "SIL @f go to los angeles" would have 3 "words" and be 100% correct, but "SIL @f go to san francisco SIL" and "@f go from los angeles SIL" would have 3 words each and be 66.7% (2/3) correct.

3. Natural language understanding success rate: the ratio of dialogue turns in which the recognized user utterances are correctly translated as input frames, to the total number of dialogue turns where this module operates.

4. Dialogue manager turn success rate: the ratio of dialogue turns in which the dialogue manager reacts correctly to the user's input frame, to the total number of dialogue turns where this module operates.

5. Oral response generator success rate: the ratio of dialogue turns in which the oral response generator generates correct responses for the frames it
6.1. DESCRIPTION OF MEASURED QUALITY PARAMETERS

receives, to the total number of dialogue turns where this module operates.

We also calculated the following parameters about the whole system:

1. Full chain success rate: the ratio of dialogue turns in which the system’s oral response to the user’s utterance is correct, to the total number of dialogue turns. The initial turn, in which the system greets the user without any previous user input, is not counted here. The final turn, which includes a call hang-up, is correct if the hang-up is a correct action to take based on the user utterance.

2. Transaction success rate: The ratio of successful calls to the total number of calls. Three measures are given, for different meanings of “successful”:
   a. The user obtains the information that she was supposed to obtain, and the dialogue finishes correctly.
   b. The user believes that she has obtained the desired information (even if this is not true), and the dialogue finishes correctly.
   c. The user obtains what she believes to be all or part of the desired information.

3. Subjective evaluation: for each call, the caller gave a subjective rating from 0 (I disagree) to 5 (I agree) to the following questions (mostly drawn from [37], but with a different mark scale):
   a. It was easy to get the information that I wanted.
   b. The system understood what I said.
   c. I found it easy to understand what the system said.
   d. I knew what I could say or do at each point in the dialogue.
   e. The system worked the way I expected it to.
   f. I would like to use this system regularly.

4. Average call duration.

The author was also a test user. To keep values fair, he did not give a subjective evaluation; objective metrics for his calls were however used. It should also be noted that most test users had a personal acquaintance with the author; therefore subjective results should be taken with a grain of salt.
6.2 Call scenarios

There were seven different call scenarios. Three of them were considerably more demanding (from the dialogue system’s point of view). Each experimenter was asked to choose one difficult scenario and two simpler scenarios, before reading them.

Each scenario had some questions that the user should ultimately be able to answer from information she got from the system. Transaction success is defined as the user being able to reply to these questions, not on the whole dialogue being error-free.

We give a translation of these call scenarios:

1. You live in Barcelona. Next Thursday, at 12:00 (noon) you have a business meeting in Zaragoza. You would like to come back home as soon as possible, but you will not be able to be back at the train station (in Zaragoza) until 17:00.

   Questions: Departure time, from Barcelona, of the train that arrives at Zaragoza closest to, but before, 12:00. Arrival time of that train. Departure time, from Zaragoza, of the first train that leaves after 17:00. Arrival time of that train.

2. You live in Girona. You would like to spend next weekend in Barcelona. However, you should be back home Sunday night, and on Friday you can’t leave the office until six o’clock. Choose a cheap ticket if possible.

   Questions: Departure date and time from Girona, and arrival date and time to Barcelona. Departure date and time, and arrival date and time, for the return train.

3. You live in Valladolid. You have some business to do in Madrid next Friday. After that, you won’t go back to Valladolid because you’d like to visit a relative in Cuenca. You should arrive at Madrid before one o’clock. You’ll spend at least two hours in Madrid. If you have some extra time, you may do some tourism, but you should be in Cuenca before dinner time. Apart from these restrictions, choose whatever you like.


4. You’d like to go from Madrid to Zaragoza tomorrow. Any time is acceptable, but you want a cheap ticket. What is the fare price (one way), departure time and arrival time?

5. What is the departure time for the Barcelona-Paris “Talgo” train next Saturday?
6.3. Results

6.3.1 Qualitative remarks

We discovered some interesting trends while monitoring the execution of the test calls:

1. People didn’t use all the meta-communication and navigation possibilities. Apparently they assumed, incorrectly, that, being a machine, the system could understand only a very limited range of sentences. For example, people didn’t realize that they could use complex time expressions, and that they could give an arrival time when asked for a time.

We believe that this is a consequence of the user’s preconceptions of what a machine can do, reinforced by the artificially-sounding synthesized voice. Perhaps we could rephrase the prompts so that they explain more clearly to the user their full range of options.

2. People did not use the negation-correction feature in explicit confirmations, which allows them to say, for example, No, I’d like to leave from New York when the system asks Would you like to leave from Denver?. Instead, they would just say No and let the system ask them the departure place. This makes dialogues longer.

We do not think that we should tell the user about this option explicitly, because this combined negation-correction answer is natural in human speech. Perhaps the problem is, again, that users think that the system is not capable of handling that.

3. People adapted their language when they realized they were talking to a machine, by simplifying it and using simpler expressions.

For example, we had a user who tried to give a full time (such as sixteen forty-two for 16:42). The system is capable of handling these expressions, but for some other reason (probably low audio quality) there was a recognition failure. The user thought that the expression was too complex and adapted to that by giving a shorter approximation (five o’clock) in the next attempt.

4. People learn how to use the system. Each experimenter made three calls, and we saw that the dialogue went much more smoothly in the third call.
than in the first one. They learn what to expect from the system, how
to speak in order to get less recognition failures, and how to guide the
dialogue towards their goal.

5. The system does not allow barge-in, and signals this by playing a short
beep when it is ready to listen. People intuitively learn to speak after the
beep after just one dialogue turn. Some users forgot to wait for the beep
on some turns, but we found that this did not hurt much the dialogue
—at most, there was a recognition failure for that turn—and the user
would quickly remember she should have waited for the beep.

6. We detected signs of fatigue in many experimenters. The reason, accord-
ing to the experimenters themselves, is that dialogues are far too long
and slow. This relates to two problems: First, the speech synthesizer
is usually too slow; we might try to make system output shorter, even
if this would make it less polite. Second, since we use a speech recogni-
tizer that doesn’t give recognition confidence values, we had to be very
conservative in our confirmation strategy and use much more explicit
confirmations than would be strictly necessary.

6.3.2 Quantitative results

In this section we give the numerical results we got from the evaluation calls.
For success/failure parameters, we give the number of successful results and
the total number of results only for the table that aggregates all tests. For
other tables, we give that as a percent ratio only.

For turn- and word-based parameters (such as speech recognizer word success
rate, or full chain turn success rate) we can calculate two different percents:
the first one (single average) is computed by dividing the total number of
successful results by the total number of results in the result set. The second
one (call average) is the average of the following values: for each turn, divide
the number of successful results in the turn by the total number of results in the
turn. We are not sure which one is more meaningful. In single average results,
each dialogue turn (or each word) counts the same; in call average results, we
compensate for the fact that successful dialogues tend to be shorter. In charts,
and in tables where we divide the samples into classes, we only give the second
percent. In the table for the whole set, we give the single average in the second
column and the call average in the third one.

In the figures, we use the following abbreviations:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td># tests</td>
<td>number of test calls used for this table</td>
</tr>
<tr>
<td>SR</td>
<td>speech recognizer turn success rate</td>
</tr>
<tr>
<td>SR word</td>
<td>speech recognizer word success rate</td>
</tr>
</tbody>
</table>
6.3. RESULTS

NLU  NLU module turn success rate
DM   dialogue manager turn success rate
ORG  ORG module turn success rate
full full chain turn success rate
t. real transaction success (a): the user really gets the information
t. user t. success (b): the user thinks she has got the information
t. part t. success (c): the user thinks she has got part of the information
t ime average call duration
s. info “it was easy to get the information that I wanted”
s. SR  “the system understood what I said”
s. TTS “I found it easy to understand what the system said”
s. guided “I knew what I could say or do at each point in the dialogue”
s. expect “the system worked the way I expected it to”
s. use  “I would like to use this system regularly”

We must note that we were only able to run a very limited number of tests, due to time and resource constraints; therefore sample noise (differences between distribution means and sample means) should not be ignored. We show 95% confidence intervals as error bars.

Figure 6.1 shows the results from all tests. We see that subjective ratings are around 3, just a little bit above the range midpoint (2.5), for all questions except questions (c) and (d). Question (c), *I found it easy to understand what the system said*, gets significantly higher answers, which shows that the TTS had a very good performance, and that the prompts could be easily understood. We also get a high mark (around 4) in question (d), *I knew what I could say or do at each point in the dialogue*, which shows that the dialogue strategy was well designed.

We get speech recognizer success rates between 84.5% and 89%. While some systems achieve even higher recognition rates, this is quite good. We further see that the full chain turn success rate is similar to the speech recognizer success rate —given that the NLU, DM, and ORG modules have almost 100% success rates, this means most whole chain failures were caused by recognition failures.

The reason why the NLU, DM and ORG modules have such high success rates compared to the speech recognizer is that the former operate very deterministically on their input (frames and recognized words); failures in these modules indicate programming mistakes, or problems in the application design. The actual errors we found were not programming mistakes but rather shortcomings in the application design that we already knew about but had decided not to solve because of the high cost of doing so compared to the potential gain.

On the other hand, failures in the speech recognition module do not (in principle) come from programming mistakes, but rather from poor input quality (noisy speech, strangely-pronounced words, uncooperative speaker, …) or
### CHAPTER 6. EVALUATION

**Figure 6.1: Results from all tests**

<table>
<thead>
<tr>
<th></th>
<th>all users</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong># tests</strong></td>
<td>27</td>
</tr>
<tr>
<td><strong>SR</strong></td>
<td>528/604</td>
</tr>
<tr>
<td><strong>SR word</strong></td>
<td>764/903</td>
</tr>
<tr>
<td><strong>NLU</strong></td>
<td>603/604</td>
</tr>
<tr>
<td><strong>DM</strong></td>
<td>598/604</td>
</tr>
<tr>
<td><strong>ORG</strong></td>
<td>622/631</td>
</tr>
<tr>
<td><strong>full</strong></td>
<td>531/604</td>
</tr>
<tr>
<td><strong>t. real</strong></td>
<td>19/27</td>
</tr>
<tr>
<td><strong>t. user</strong></td>
<td>20/27</td>
</tr>
<tr>
<td><strong>t. part</strong></td>
<td>25/27</td>
</tr>
<tr>
<td><strong>time</strong></td>
<td>297.1s</td>
</tr>
<tr>
<td><strong>s. info</strong></td>
<td>3.0/5.0</td>
</tr>
<tr>
<td><strong>s. SR</strong></td>
<td>3.6/5.0</td>
</tr>
<tr>
<td><strong>s. TTS</strong></td>
<td>4.4/5.0</td>
</tr>
<tr>
<td><strong>s. guided</strong></td>
<td>4.0/5.0</td>
</tr>
<tr>
<td><strong>s. expect</strong></td>
<td>3.3/5.0</td>
</tr>
<tr>
<td><strong>s. use</strong></td>
<td>2.9/5.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>all users</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SR word</strong></td>
<td></td>
</tr>
<tr>
<td><strong>NLU</strong></td>
<td></td>
</tr>
<tr>
<td><strong>DM</strong></td>
<td></td>
</tr>
<tr>
<td><strong>ORG</strong></td>
<td></td>
</tr>
<tr>
<td><strong>full</strong></td>
<td></td>
</tr>
<tr>
<td><strong>t. real</strong></td>
<td></td>
</tr>
<tr>
<td><strong>t. user</strong></td>
<td></td>
</tr>
<tr>
<td><strong>t. part</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Time**

<table>
<thead>
<tr>
<th></th>
<th>all users</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>s. info</strong></td>
<td></td>
</tr>
<tr>
<td><strong>s. SR</strong></td>
<td></td>
</tr>
<tr>
<td><strong>s. TTS</strong></td>
<td></td>
</tr>
<tr>
<td><strong>s. guided</strong></td>
<td></td>
</tr>
<tr>
<td><strong>s. expect</strong></td>
<td></td>
</tr>
<tr>
<td><strong>s. use</strong></td>
<td></td>
</tr>
</tbody>
</table>
because the language model does not cover some specific input (this is not necessarily a problem with the language model—we knew from the start that we would try to understand only the most common expressions).

Finally, we have transaction success rates around 70.5%–74% (92.5% for partial transaction successes), which is also good.

We were surprised to have such high figures given the lower subjective ratings. Apparently, while users complain that it is difficult for them to obtain the information they needed, they still manage to get it. From this we think that this system may be useful for situations where remote voice access is a requirement, but probably not where it is only an option; for example, for blind people, where Internet access is not available, and away from a train station where posted schedules could be checked.

We divided the result set into different classes, to evaluate the effect of several factors on the system performance.

**Effect of user experience on speech technologies**

First, we divided between users with experience on speech technologies and users without. We expected that users having worked in that field would know, from their previous reading, how to interact with a machine-driven dialogue system, and that this would result in increased success rates.

However, as we can see in figure 6.2, this is not the case. Actually, the reverse is true. On all subjective ratings, the system gets lower marks from users working in the field that from the other users, and lower transaction success rates. We also do not see any substantial improvement on the speech recognition success rate, contrary to what would seem natural. It may all be an effect of the small sample size.

**Effect of user experience with the system**

During the tests, we got the impression that testers learned to use the system very quickly and adapted their behavior to match the system in order to complete their tasks more quickly. We analyzed if this impression was actually true by dividing the sample set in three classes: the first time that the user used the system, the second time, and all the following times. This is shown in figure 6.3. Some users (for example, the author) had all tests classified as “following times” because they had tried the system before.

We see that there is a consistent improvement of subjective ratings and transaction success rates as users become more experienced. We also see how the
### Figure 6.2: Effect of user experience on speech technologies

<table>
<thead>
<tr>
<th></th>
<th>with experience</th>
<th>without experience</th>
</tr>
</thead>
<tbody>
<tr>
<td># tests</td>
<td>21</td>
<td>6</td>
</tr>
<tr>
<td>SR</td>
<td>89.4%</td>
<td>87.6%</td>
</tr>
<tr>
<td>SR word</td>
<td>86.4%</td>
<td>87.2%</td>
</tr>
<tr>
<td>NLU</td>
<td>99.7%</td>
<td>100.0%</td>
</tr>
<tr>
<td>DM</td>
<td>99.9%</td>
<td>99.2%</td>
</tr>
<tr>
<td>ORG</td>
<td>97.4%</td>
<td>100.0%</td>
</tr>
<tr>
<td>full</td>
<td>89.8%</td>
<td>88.4%</td>
</tr>
<tr>
<td>t. real</td>
<td>66.7%</td>
<td>83.3%</td>
</tr>
<tr>
<td>t. user</td>
<td>71.4%</td>
<td>83.3%</td>
</tr>
<tr>
<td>t. part</td>
<td>95.2%</td>
<td>83.3%</td>
</tr>
<tr>
<td>time</td>
<td>264.1s</td>
<td>412.7s</td>
</tr>
<tr>
<td>s. info</td>
<td>2.8/5.0</td>
<td>3.5/5.0</td>
</tr>
<tr>
<td>s. SR</td>
<td>3.7/5.0</td>
<td>3.5/5.0</td>
</tr>
<tr>
<td>s. TTS</td>
<td>4.2/5.0</td>
<td>4.8/5.0</td>
</tr>
<tr>
<td>s. guided</td>
<td>3.8/5.0</td>
<td>4.5/5.0</td>
</tr>
<tr>
<td>s. expect</td>
<td>3.2/5.0</td>
<td>3.5/5.0</td>
</tr>
<tr>
<td>s. use</td>
<td>2.7/5.0</td>
<td>3.5/5.0</td>
</tr>
</tbody>
</table>
### 6.3. RESULTS

<table>
<thead>
<tr>
<th></th>
<th>first call</th>
<th>second call</th>
<th>following calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong># tests</strong></td>
<td>7</td>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td><strong>SR</strong></td>
<td>88.1%</td>
<td>89.6%</td>
<td>89.2%</td>
</tr>
<tr>
<td><strong>SR word</strong></td>
<td>88.0%</td>
<td>83.6%</td>
<td>87.4%</td>
</tr>
<tr>
<td><strong>NLU</strong></td>
<td>100.0%</td>
<td>100.0%</td>
<td>99.5%</td>
</tr>
<tr>
<td><strong>DM</strong></td>
<td>99.0%</td>
<td>99.2%</td>
<td>99.5%</td>
</tr>
<tr>
<td><strong>ORG</strong></td>
<td>94.5%</td>
<td>100.0%</td>
<td>98.8%</td>
</tr>
<tr>
<td><strong>full</strong></td>
<td>89.0%</td>
<td>90.4%</td>
<td>89.2%</td>
</tr>
<tr>
<td><strong>t. real</strong></td>
<td>42.9%</td>
<td>71.4%</td>
<td>84.6%</td>
</tr>
<tr>
<td><strong>t. user</strong></td>
<td>42.9%</td>
<td>85.7%</td>
<td>84.6%</td>
</tr>
<tr>
<td><strong>t. part</strong></td>
<td>85.7%</td>
<td>100.0%</td>
<td>92.3%</td>
</tr>
<tr>
<td><strong>time</strong></td>
<td>440.7s</td>
<td>299.3s</td>
<td>218.7s</td>
</tr>
</tbody>
</table>

![Graph showing performance metrics](image)

**Figure 6.3:** Effect of user experience with the system
average call duration gets shorter and shorter as the user learns how to guide
the system towards her goal.

On the other hand, we do not see any improvement in any of the individual
modules or in the full chain success rate. This means that users become more
efficient by learning to use certain replies and dialogue paths, but not by learn-
ing to speak in a more understandable manner or use expressions that the
system understands better.

We should however note that this data may be contaminated by the fact that
most “first times” were hard scenarios while most “second times” and “follow-
ing times” were easy scenarios. While the actual figures may be distorted, we
believe that the overall trend is still true, for two reasons: First, not all hard
scenario calls were “first times”. Second, if the only reason for the improve-
ment were the different scenario difficulty, we should not see any improvement
between the second call and the following calls, but we do see some.

Effect of scenario difficulty

We also decided to directly analyze the effect of scenario difficulty, by separating
calls with hard scenarios (the first three scenarios) from calls with easy scenarios
(the last four). This is shown in figure 6.4.

As before, these figures may be contaminated by the fact that hard scenario
calls were usually performed first in the run of three calls that each tester made.
Surprisingly however, we find that there is not any significant improvement in
the transaction success rate for easy scenarios (compared to difficult scenarios).
There is some improvement in the overall subjective rating. And calls are much
shorter, but this is simply because complex tasks take more time to complete,
even in human-human dialogues.

Effect of ambient noise

As we were running the tests, we realized that we were in a quite noisy envi-
ronment. There were five computers in the room, and people talking in the
background. Even if that level of noise was not bothersome for humans, we
thought that it might have an adverse effect on system performance. From a
certain moment on, we decided to move our testing to another room, which
was quieter than the original one (but in no way silent). Figure 6.5 compares
both situations.

We can easily see that there was an increase in subjective ratings and whole
transaction success rates as we moved to the quieter room. We would expect
that the quieter environment would give higher recognition rates, which would
lead to a better overall performance. However, we do not see any significant
### 6.3. Results

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>Easy</th>
<th>Hard</th>
</tr>
</thead>
<tbody>
<tr>
<td># Tests</td>
<td>19</td>
<td>8</td>
</tr>
<tr>
<td>SR</td>
<td>89.5%</td>
<td>87.8%</td>
</tr>
<tr>
<td>SR word</td>
<td>87.7%</td>
<td>83.9%</td>
</tr>
<tr>
<td>NLU</td>
<td>99.6%</td>
<td>100.0%</td>
</tr>
<tr>
<td>DM</td>
<td>99.5%</td>
<td>98.8%</td>
</tr>
<tr>
<td>ORG</td>
<td>97.6%</td>
<td>99.1%</td>
</tr>
<tr>
<td>Full</td>
<td>89.8%</td>
<td>88.6%</td>
</tr>
<tr>
<td>Time real</td>
<td>68.4%</td>
<td>75.0%</td>
</tr>
<tr>
<td>Time user</td>
<td>73.7%</td>
<td>75.0%</td>
</tr>
<tr>
<td>Time part</td>
<td>94.7%</td>
<td>87.5%</td>
</tr>
<tr>
<td>Time</td>
<td>197.8s</td>
<td>533.0s</td>
</tr>
</tbody>
</table>

- **s. info** 3.2/5.0 2.6/5.0
- **s. SR** 3.8/5.0 3.3/5.0
- **s. TTS** 4.3/5.0 4.6/5.0
- **s. guided** 3.9/5.0 4.1/5.0
- **s. expect** 3.4/5.0 3.1/5.0
- **s. use** 3.2/5.0 2.1/5.0

![Graph showing the effect of scenario difficulty](image)

**Figure 6.4: Effect of scenario difficulty**
increase in recognition rates.

Given that the speech recognition does not get any better, we are not sure what the reason for the improvement of the other parameters might be. Perhaps the quieter room made the experimenters less stressed, or maybe the kind of recognition errors made in the noisy situation and those made in the quiet situation are of a different kinds, the former having a larger impact on the dialogue.

![Graph showing effect of ambient noise](image)

**Figure 6.5: Effect of ambient noise**

**Correlation of results to dialogue success**

Finally, we divided the tests depending on whether they were successful or unsuccessful dialogues (as given by user transaction success). This may seem nonsense (after all, “of course successful dialogues have better results!”) but
6.3. RESULTS

we wanted to do it anyway just to be sure and, most importantly, to see how much better there results are. The comparison is shown in figure 6.6.

We of course get mostly what we expected, better results in successful dialogues, but we also get some interesting data that we hadn’t anticipated:

First, the users gave almost the same average answer to the questions “I would like to use this system regularly” and “I knew what I could say or do at each point in the dialogue”. We also get almost the same answer to the question “I found it easy to understand what the system said”, but we expected that, since we use the same speech synthesizer and the same system prompts. We are surprised that the overall user satisfaction, as measured by the question “I would like to use this system regularly”, is not affected by dialogue success.

![Figure 6.6: Correlation of results to dialogue success](image)

We also analyzed whether the experimenter’s gender had any effect on the results. Unfortunately, our sample size is too small to be able to infer any
conclusions from that.

6.3.3 User comments

The users were asked to give their comments on the system after each call. We were especially interested in complaints, which may help us improve the system:

1. A user complained that it was easy to get schedule information, but much harder to get additional information such as prices or train types.

2. A user actually realized that he was learning to use the system.

3. Another user complained that the system talked too fast for him to write down all the information he needed.
   
   We think this is true. At the same time, people felt that the system talked too slowly when it was not giving them information they wanted. There are two problems here:
   
   First of all, hardware limitations make it impossible for us to make the system talk faster. We also tried to make it talk more slowly on selected sentences, but it did not work very well. The real problem, however, is that the system is not able to detect when the user is interested about what it is saying and when she is not. In many cases exactly the same sentence was “interesting” or “uninteresting” in different situations. We do not see a way to solve this problem which does not involve some more complex analysis of the speaker (for example, detecting emotions from the user’s voice).

4. Finally, a user complained about the dialogue structure not being flexible enough. The system prompts were also too static for him.

   This could be partially solved by improving a little bit the dialogue strategy and the oral responses. However, we believe that these will never be complete solutions. The only way to get a flexible enough system is possibly to make it user-driven instead of machine-driven.

6.3.4 Comparison to other systems

As said before, it is unfortunately not possible to compare this system to the Basurde system.

For reference, we give here some performance ratings of other dialogue systems. One of the problems we found in this comparison was that not two projects use the same kind of performance metrics, but still we tried to present some comparable measures.
6.3. RESULTS

1. The DEMON system [28], a mixed-initiative dialogue system for accessing railway information. They report a 67% transaction success rate, a 12% failure rate, and a 20% of “inappropriate information given”, where the system gave the wrong information to the user (we considered this a dialogue failure in Basurde[lite]).

2. The LIMSI ARISE system [43], also a mixed-initiative system for railway information. For recognizer word error rates similar to those found in Basurde[lite], they report transaction success rates of around 80%.

3. The CU Communicator system [37], a mixed-initiative system for travel information. They report transaction success rates of 77.8%, and also give subjective metrics similar to those used in Basurde[lite]. The average results for all DARPA Communicator systems is also given. We converted the subjective results to the [0,5] scale used for Basurde[lite].

4. The ACCeSS system [20], a machine-driven system. They report dialogue success rates of 91%, and provide measures of whole-sentence recognition rate, word recognition rate, and concept accuracy rate (in the input side).

5. The LIMSI RAILTEL system [7], a mostly machine-driven system for railway information. They do not give transaction success measures, but give concept recognition rates and full chain turn success rates.

Table 6.1 summarizes these figures. For the Basurde[lite] system, we give both the real and the user transaction success rates.

We can see that the Basurde[lite] rates fairly among these other systems.

<table>
<thead>
<tr>
<th>Basurde [lite]</th>
<th>DEMON</th>
<th>LIMSI ARISE</th>
<th>CU Communicator</th>
<th>DARPA</th>
<th>ACCeSS</th>
<th>LIMSI RAILTEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jul'01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t. real/usr</td>
<td>70.4%</td>
<td>67%</td>
<td>≈60%</td>
<td>77.8%</td>
<td>62.0%</td>
<td>90.8%</td>
</tr>
<tr>
<td>full</td>
<td>89.5%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>59.5%</td>
</tr>
<tr>
<td>SR</td>
<td>89.0%</td>
<td></td>
<td></td>
<td></td>
<td>≈75%</td>
<td>78.4%</td>
</tr>
<tr>
<td>s. info</td>
<td>3.0</td>
<td></td>
<td>3.6</td>
<td>2.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s. TTS</td>
<td>4.4</td>
<td></td>
<td>4.4</td>
<td>3.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s. guided</td>
<td>4.0</td>
<td></td>
<td>4.0</td>
<td>3.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s. expect</td>
<td>3.3</td>
<td></td>
<td>3.4</td>
<td>2.6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Comparison to other systems
6.4 Two example dialogues

For illustration, here is the transcription of two real dialogues held by testers and the Basurde[lite] system.

The parts of the user input that were ignored (not just incorrectly recognized) by the speech recognizer are shown in italics. User speech is preceded by “U”; system speech is preceded by “S”.

S[01]—Buenas tardes. Le atiende el servicio de información sobre trenes regionales y de largo recorrido. ¿Quiere que le explique cómo funciona el sistema?
U[01]—No.
S[02]—¿Desea salir de Barcelona?
U[02]—Sí.
S[03]—¿Cuál es su destino?
U[03]—Valencia.
S[04]—A Valencia. ¿Qué día desea salir?
U[04]—Quiero saber el precio.
S[05]—¿Desea salir el trece?
U[05]—No, el fin de semana.
S[06]—¿Desea salir entre el veinte de julio y el veintidós de julio?
U[06]—Sí.
S[07]—Un momento, por favor.
S[07]—Hay dieciséis trenes de Barcelona a Valencia que salen el viernes veinte de julio. Hay trece trenes de Barcelona a Valencia que salen el sábado veintiuno de julio. Hay catorce trenes de Barcelona a Valencia que salen el domingo veintidós de julio. ¿Qué día desea salir?
S[08]—Me escucha? ¿Qué día desea salir?

Good afternoon, here is the information service for regional and long-distance trains. Would you like me to explain how to use this system?
No.
Do you wish to leave from Barcelona?
Yes.
Where are you going to?
Valencia.
To Valencia. On what date would you like to leave?
I’d like to know the price.
Would you like to leave on the 13th?
No, a weekend.
Would you like to leave between July 20th and July 22nd? [these were the dates for the following weekend]
Yes.
One moment, please.
There are 16 trains from Barcelona to Valencia leaving Friday, July 20th. There are 13 trains from Barcelona to Valencia leaving Saturday, July 21st. There are 14 trains from Barcelona to Valencia leaving Sunday, July 22nd. On what date would you like to leave?
Hello? On what date would you like to leave?
6.4. TWO EXAMPLE DIALOGUES

U[08]—El veintiuno.
S[09]—¿Desea salir el veintiuno?

U[09]—Sí.
S[10]—Hay trece trenes de Barcelona a Valencia que salen el veintiuno. El primero sale a las siete y llega a las nueve y cincuenta. El último sale a las veintiuna y llega a las cero y veinticinco. Por favor, escoja cómo seleccionar un tren: según su horario, los servicios que ofrece, el tipo de tren, o el más barato. Diga no, si no le interesa ninguno.

U[10]—Horarios.
S[11]—¿A qué hora desea salir?

U[11]—Temprano, hacia las ocho.
S[12]—¿Desea salir entre las cuatro de la madrugada y las diez de la mañana?
U[12]—Mejor entre las ocho y las diez.
S[13]—¿Desea salir entre las once de la mañana y las ocho y veinte de la tarde?
U[13]—No, no, entre las ocho de la mañana y las diez de la mañana.
S[14]—¿Desea salir entre las ocho de la mañana y las siete de la mañana?
U[14]—No, entre las ocho y las [end of sentence cut by audio server]
S[15]—¿A qué hora desea salir?

U[15]—A las nueve.
S[16]—¿Salir a las nueve de la mañana o de la noche?
U[16]—De la mañana.
S[17]—¿Desea salir a las nueve de la mañana?
U[17]—Sí.

The 21st.
Would you like to leave on the 21st?
Yes.

There are 13 trains from Barcelona to Valencia leaving on the 21st. The first one leaves at 07:00 and arrives at 09:50. The last one leaves at 21:00 and arrives at 00:25. Please select one way of choosing a train: by its schedule, the services offered on it, the kind of train, or the cheapest one. Say no, if you are not interested in any train.

Schedule.
At what time would you like to leave?
Soon, around eight.

Would you like to leave between four in the morning and ten in the morning?
Rather between eight and ten.

Would you like to leave between eleven in the morning and eight twenty in the afternoon?
No, no, between eight in the morning and ten in the morning.

Would you like to leave between eight in the morning and seven in the morning?
No, between eight and . . .

At what time would you like to leave?
At nine.
Leave at nine in the morning or in the evening?
In the morning.

Would you like to leave at nine in the morning?
Yes.
S[18]—Hay un tren de Barcelona a Valencia que sale el sábado veintiuno de julio a las nueve y que llega a las once y cincuenta y cuatro. ¿Quieres más información de este tren? Por favor, diga sí o no.

U[18]—Sí, sí.

S[19]—El tren va de Barcelona a Valencia, sale el sábado veintiuno de julio a las nueve y llega a las once y cincuenta y cuatro. Es un tren Euromed. Cuenta con cafetería, teléfono, video y prensa. En clase preferente, el billete vale ocho mil pesetas para sólo ida, y doce mil ochocientos para ida y vuelta. En clase turista, el billete vale cinco mil doscientas pesetas para sólo ida, y ocho mil trescientas veinte para ida y vuelta. ¿Quiere información del anterior o siguiente tren?

U[19]—Sí, del anterior.

S[20]—¿Quiere información del anterior o siguiente tren?

U[20]—Sí, quiero información del tren anterior.

S[21]—¿Quiere información del anterior o siguiente tren?

U[21]—Sí, del anterior.

S[22]—El tren va de Barcelona a Valencia, sale el sábado veintiuno de julio a las ocho y llega a las once y veintitrés. ¿Quiere más información de este tren? Por favor, diga sí o no.

U[22]—Sí.

There is a train from Barcelona to Valencia leaving on Saturday, July 21st at 09:00, arriving at 11:54. Do you need more information about this train? Please answer yes or no.

Yes, yes.

The train travels from Barcelona to Valencia; it leaves on Saturday, July 21st at 09:00 and arrives at 11:54. It's an “Euromed” train. It has a cafeteria, telephone, movies and newspapers. In first class, a one-way ticket costs 8000 peseta, and a return ticket costs 12800 peseta. In economy class, a one-way ticket costs 5200 peseta, and a return ticket costs 8320 peseta. Do you need information about the previous or next train?

Yes, about the previous one.

Do you need information about the previous or next train?

Yes, I want information about the previous train.

Do you need information about the previous or next train?

Yes, about the previous one.

The train travels from Barcelona to Valencia; it leaves on Saturday, July 21st at 08:00, arriving at 11:23. Do you need more information about this train? Please answer yes or no.

Yes.
6.4. TWO EXAMPLE DIALOGUES

S[23]—El tren va de Barcelona a Valencia, sale el sábado veintiuno de julio a las ocho y llega a las once y veintitrés. Cuenta con cafetería. En clase preferente, el billete vale cuatro mil ochocientas pesetas para sólo ida, y siete mil seiscientas ochenta para ida y vuelta. En clase turista, el billete vale tres mil seiscientas pesetas para sólo ida, y cinco mil setecientas sesenta para ida y vuelta. ¿Quiere información del anterior o siguiente tren?

U[23]—No.

S[24]—¿Desea información sobre otros trenes? Por favor, responda sí o no.

U[24]—No.

S[25]—Gracias por usar este servicio, y buen viaje.

The train travels from Barcelona to Valencia; it leaves on Saturday, July 21st at 08:00 and arrives at 11:24. It has a cafeteria. In first class, a one-way ticket costs 4000 pesetas, and a return ticket costs 7780 pesetas. In economy class, a one-way ticket costs 3600 pesetas, and a return ticket costs 5760 pesetas. Do you need information about the previous or next train?

No.

Would you like information about other trains? Please answer yes or no.

No.

Thank you for using this service, and have a nice trip.

S[01]—Buenas tardes. Le atiende el servicio de información sobre trenes regionales y de largo recorrido. ¿Desea instrucciones sobre el uso del sistema?

U[01]—No.

S[02]—¿Desea salir de Barcelona?

U[02]—Sí.

S[03]—¿Cuál es su destino?

U[03]—Lleida.

S[04]—A Lleida. ¿Qué día desea salir?

U[04]—El próximo miércoles.

S[05]—¿Desea salir el trece?

U[05]—No.

S[06]—¿Qué día desea salir?

Good afternoon, here is the information service for regional and long-distance trains. Would you like me to explain how to use this system?

No.

Do you wish to leave from Barcelona?

Yes.

Where are you going to?

Lleida.

To Lleida. On what date would you like to leave?

Next Wednesday.

Would you like to leave on the 13th?

No.

On what date would you like to leave?
U[06]—El miércoles.
S[07]—¿Desea salir el miércoles?

U[07]—Sí.
S[08]—Un momento, por favor.
S[08]—Hay diecinueve trenes de Barcelona a Lleida que salen el miércoles. El primero sale a las seis y cincuenta y dos y llega a las nueve y cincuenta. El último sale a las veintitrés y llega a la una y veintitrés. Por favor, escoja cómo seleccionar un tren: según su horario, los servicios que ofrece, el tipo de tren, o el más barato. Diga no, si no le interesa ninguno.

U[08]—Por los servicios que ofrece.
S[09]—Hay trenes con cafetería, vídeo, prensa, cama y restaurante. ¿Qué servicio le interesa?

U[09]—Restaurante.
S[10]—¿Un tren con restaurante?
U[10]—Sí.
S[11]—Hay dos trenes con restaurante, de Barcelona a Lleida que salen el miércoles. El primero es un tren hotel, sale a las veintidós y llega a las cero y diez. El segundo es un tren hotel, sale a las veintitrés y llega a la una y veintitrés. ¿Quiere información del primer o segundo tren?


...
Chapter 7

Conclusion

7.1 Concluding remarks

Since the Basurde system was not complete at the time of writing this report, we cannot unfortunately compare both systems and see if using a machine-driven strategy gives higher quality (because of less recognition failures) or lower quality (because of a less flexible dialogue). We obtained performance measures for Basurde[lite] and compared them to other dialogue systems. We see that the system gets good performance measures.

We managed to fulfill the other major goal, to develop a dialogue system for accessing the same information that Basurde does in much less time than it would take to develop Basurde.

The whole system runs very fast except for the speech recognition module. On a computer with a Intel Pentium II CPU at 350 MHz, with 384 Mbyte of RAM, recognition of simple sentences takes between 1 and 3 seconds. Subjective impressions show that this is twice slower than what would be adequate. Since 1 GHz i386 CPUs are already available, we think that this will not be a problem on a production system equipped with the most modern hardware.

The decision of using a full programming language as the language for configuration files proved to be a good one: In this document we only describe the final version of the system, but before that one we produced a partial ORG module using a limited configuration language, and a full system using a different dialogue strategy. We found out that we could produce a much more powerful ORG module with Tcl as the configuration language than without, and we could do it faster. We also found out that changing the dialogue strategy was easy with that scheme.

Users found the system usable for simple tasks, but a little bit less apt for more difficult tasks. We think that machine-driven strategies are not flexible enough to allow the user to execute complex tasks, and that for these complex
tasks these strategies should not be used. On the other hand, simple tasks, especially if there are only a few of them which can be analyzed in advance, do not require much flexibility, and restricted-vocabulary, machine-driven systems are adequate.

Our idea that using many small grammars would increase the recognition success rate did not, apparently, have an effect so large as we expected. On the other hand, we have not made tests combining all small grammars, and we do not know what the results would be in that case. A reason could be that we have used probability-less grammars, that is, every possible input string has the same probability of being said, as far as the grammar is concerned (there are probability restrictions at the phonetic level, however). Maybe including probabilities in the input grammars would improve the system. However we do not have the resources (large corpora) needed to assign realistic probabilities to strings.

In conclusion, this is a fully operational system capable of handling low-complexity dialogues about railway schedules. It has a good performance given its complexity and development time limitations.

### 7.2 Lessons learned

During the design and implementation of the Basurde[lite] system, and especially during its evaluation phase, we gathered some impressions about these systems that may be useful for the future development of other machine-driven dialogue systems:

We found that implementing a domain-specific language, and doing this through a language more suited for rapid prototyping than C or C++ (which was Tel in our project) was a key factor in getting the project done on time and, more importantly, in letting us refine the application design as we discovered, through testing, parts that could be improved.

However, we think that the decision not to use higher-level languages such as Haskell or Scheme should be reevaluated. We saw that programming some application-definition files would have been easier with more powerful languages.

We discovered that it was useless to make the system offer the user a “usage guide”. Very rarely would the users ask for it, and absolutely never did they actually use those instructions. But we saw that we could “train” the user how to use the system by means of properly designed system prompts. We saw that the precise wording of system prompts was crucial in determining the kind of user answer we would get.

We also noticed that keeping dialogues short and fast-paced is important to
7.3. FUTURE WORK

user satisfaction (though not really for transaction success). Using a speech recognizer that can give recognition confidence values, and using them to reduce the number of confirmation requests will prove, we believe, very effective in reducing dialogue length.

Allowing user barge-in would enable us to further reduce the utilization of confirmation requests, thus making dialogues even shorter.

We saw that users had no trouble at all navigating through the slots in a task form (for example, giving the departure place, arrival place, date, and time), but had more trouble when they wanted the system to change task (from schedule information to pricing information, or to train availability). Of course the system was not designed with such a clear task division in mind, but after the evaluation phase we think that we should let the user explicitly choose her current task (for example, we could give, right at the beginning, a question such as "What do you want to know: schedules, prices, or you want to find a train with some specific schedule, services or type?").

Finally, we think that it is important to have a way of knowing which information is interesting to the user and which is not, because this would allow the system to say that information differently (for example, more slowly and in more detail if the user is interested on it).

7.3 Future work

This system cannot be extended much further without foregoing one of the main goals of this project: to develop a simplified version of a complex dialogue system, and to compare both.

However there are some ways in which this work could be continued without adding significant complexity:

1. Use probabilities in the language model grammars. Currently, all input strings are equiprobable to the speech recognizer.

2. Allow user barge-in. This was not implemented mainly because of hardware limitations. However it would make user corrections easier without making the system much more complex.

3. Doing more extensive evaluations, and using them to iteratively refine the input grammars and the dialogue strategy.

4. Find some simple method that would give us a rough measure of the interest of the user in a particular piece of information. The system would give that information in different ways depending on that measure.

5. Modify the speech recognizer so that it gives a recognition confidence measure. The NLU and DM modules are already prepared to handle
recognition confidences (though this feature has not been tested). Using confidence measures would allow us to be much more aggressive in the confirmation strategy, to ask for confirmation only the data which we received with low confidence. This would make dialogues shorter and with less uncorrected recognition mistakes.

all the modern things
like cars and such
have always existed
they've just been waiting in a mountain
for the right moment
listening to the irritating noises
of dinosaurs and people
dabbling outside

all the modern things
have always existed
they've just been waiting
to come out
and multiply
and take over
it's their turn now . . .

—Björk and G. Massey, "The Modern Things"
Part III

Appendices
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


