

# Steps towards Multi-Party Dialogue Management



# Steps towards Multi-Party Dialogue Management

Fredrik Kronlid



The Graduate School of Language Technology



UNIVERSITY OF GOTHEBURG

2008

For my family

Dissertation for the degree of Doctor of Philosophy in Linguistics  
University of Gothenburg 2008-04-12

Edition for defense

© Fredrik Kronlid, 2008

Printed by Reprocentralen, Humanistiska fakulteten,  
University of Gothenburg, 2008

ISBN 978-91-628-7431-5

Typeset by the author, using L<sup>A</sup>T<sub>E</sub>X.

Distribution:

Department of linguistics, University of Gothenburg,  
Box 200, 405 30 Göteborg, Sweden

## Steps towards Multi-Party Dialogue Management

### Abstract

The aim of this work is to enable the construction of artificial conversational agents capable of working in a multi-party setting. By an artificial conversational agent we mean a software agent (possibly with embodiment, personality etc.), capable of engaging in conversation with other conversational agents (human or artificial). By a multi-party setting we mean that more than two agents are engaged in the communication or dialogue.

Components designed for two-party dialogue management including sequencing and turn management cannot, in general, handle multi-party dialogue phenomena, such as keeping track of the obligations regarding turn management and the rights and obligations to address an issue.

We present an SCXML implementation of Issue-Based Dialogue Management, scaled up to handle more than two participants. The QUD-based dialogue manager makes a distinction between *distributive* questions (to be answered individually by each addressee) and *collective* questions (to be answered by the addressees collectively), in order to provide the right kind of grounding behaviour and in order to manage correctly the obligations put on the participants of the dialogue. The dialogue manager also features an SCXML implementation of the SSJ turn management model.

We find that it is possible to scale up the Issue-Based Dialogue Management model from two-party to multi-party dialogues, given that distributive and collective readings of questions can be identified.

KEY WORDS: SCXML, multi-party, multi-party dialogue, dialogue management, turn management, QUD, issue-based dialogue management

The thesis is written in English.



## Acknowledgements

During the time that I have been a PhD student, I have taken comfort in the thought that getting the PhD done, completing a thesis and defending it – it's simply not going to happen! It was not until three months before my funding would end, that I realised that it *is* actually going to happen! It was an odd feeling – somewhere between happiness, vertigo and desperation.

There are people who have believed in me, even when I have not done so myself. First of all my main supervisor, Torbjörn Lager, who has pushed me, encouraged me and supported me during these years. I would not have done it without you. There is also my assistant supervisor, Staffan Larsson, who has been very important in many ways in the process of writing this thesis. Probably more than you think, Staffan. And without Robin Cooper, my supervisor during the first years, there would have been no thesis. Thank you Robin, for your valuable feedback on the thesis manuscript.

The support from my family has been very necessary and unconditional. Thank you, Anna, for the support (in every possible sense of the word). My daughter Elin. My son Nils. Karin and Jörgen, Lotta, Christel, Sture, Ove, Eva, Roger, Linnéa and Daniel. My friends – Andréas, Johanna, Pontus, Tobias, Björn, Stina, Andreas, Pelle.

My fellow PhD students at GSLT and at the Department of Linguistics – Stina, Ellen, Magnus G, Leif, Pontus, Håkan, Magnus S, Sofia, Markus, Ylva, Peter (not at the department, but almost), Robert (not a PhD student, but almost). It was important for me that you were there.

Of course, I need to thank all the people involved in GSLT for providing such a stimulating setting. Also, thank you everybody at the Department of Linguistics and the people taking part in the various language technology groups at University of Gothenburg – GoDaG, the Dialogue Lab, CLT etc.

Thank you Jonathan Ginzburg for kindly answering my e-mail questions and giving feedback on the manuscript. And thank you, everybody who was at Decalog in Rovereto, for a well needed injection of inspiration and confidence.



# Contents

|  |  |           |
|--|--|-----------|
| <b>1</b>                                 | <b>Introduction</b>  | <b>1</b>  |
| 1.1                                      | Purpose . . . . .  | 1         |
| 1.2                                      | A simple Dialogue Manager . . . . .                              | 2         |
| 1.3                                      | From 2-party to N-party dialogue . . . . .                       | 4         |
| 1.3.1                                    | First attempt . . . . .  | 4         |
| 1.3.2                                    | Some problematic cases . . . . .                                 | 6         |
| 1.3.3                                    | Some solutions . . . . .   | 8         |
| 1.4                                      | Discussion . . . . .   | 12        |
| <b>2</b>                                 | <b>IBDM in SCXML</b>   | <b>15</b> |
| 2.1                                      | SCXML = State Chart XML . . . . .                                | 15        |
| 2.2                                      | The ISU Approach to Dialogue Modelling . . . . .                 | 17        |
| 2.3                                      | Issue Based Dialogue Management . . . . .                        | 19        |
| 2.3.1                                    | Ginzburg's DGB and QUD . . . . .                                 | 19        |
| Structure of the common ground . . . . . | 19   |           |
| Utterance Processing Protocols . . . . . | 19   |           |
| 2.3.2                                    | Larsson's IBDM . . . . .   | 20        |
| IBiS1 . . . . .                          | 21   |           |
| IBiS2 . . . . .                          | 22   |           |
| IBiS3 and IBiS4 . . . . .                | 22   |           |
| 2.4                                      | Implementing IBiS1 in SCXML . . . . .                            | 23        |
| 2.4.1                                    | A simplified semantics . . . . .                                 | 23        |
| 2.4.2                                    | Implementing ISU Dialogue Management . . . . .                   | 23        |
| 2.4.3                                    | Implementing Information States as SCXML<br>datamodels . . . . . | 26        |
| 2.4.4                                    | Implementing dialogue moves as SCXML events . . . . .            | 27        |
| 2.4.5                                    | Implementing update rules as transitions . . . . .               | 27        |
| 2.4.6                                    | Implementing the update algorithm as a statechart . . . . .      | 27        |
| 2.4.7                                    | Implementing the DME as a statechart . . . . .                   | 34        |
| 2.4.8                                    | Modules for interpretation and generation . . . . .              | 35        |
| 2.4.9                                    | The dialogue system statechart . . . . .                         | 35        |
| 2.4.10                                   | From TrindiKit to SCXML: a summary . . . . .                     | 37        |
| 2.4.11                                   | From DIPPER to SCXML . . . . .                                   | 37        |

|          |  |           |
|----------|--|-----------|
| 2.5      | Trace of a (scripted) run of SCXML-IBiS1 . . . . .     | 40        |
| 2.5.1    | An SCXML Implementation . . . . .                      | 40        |
| 2.5.2    | The SCXML Driver . . . . .                             | 40        |
| 2.5.3    | The traces . . . . .                                   | 40        |
|          | Loading a plan to answer a question . . . . .          | 41        |
|          | Question – Answer . . . . .                            | 44        |
| 2.6      | Summary . . . . .                                      | 48        |
| <b>3</b> | <b>Turn Management</b>                                 | <b>51</b> |
| 3.1      | The Need for a Turn Management System . . . . .        | 51        |
| 3.2      | Turn Manager Requirements . . . . .                    | 52        |
| 3.3      | A Two-party Turn Manager . . . . .                     | 54        |
| 3.3.1    | Overlaps . . . . .                                     | 54        |
| 3.3.2    | TRP management . . . . .                               | 55        |
|          | Free TRPs . . . . .                                    | 56        |
|          | This agent's TRPs . . . . .                            | 56        |
|          | Other agents' TRPs . . . . .                           | 57        |
| 3.3.3    | Making it complete . . . . .                           | 57        |
| 3.4      | A Multi-party Turn Manager . . . . .                   | 59        |
| 3.4.1    | Output requirements . . . . .                          | 59        |
| 3.4.2    | Architecture . . . . .                                 | 59        |
| 3.4.3    | The Outside Chart . . . . .                            | 60        |
| 3.4.4    | The Inside Chart . . . . .                             | 61        |
|          | Overlap resolution . . . . .                           | 65        |
| 3.4.5    | The TRP Chart . . . . .                                | 65        |
| 3.4.6    | A Note on Projectability . . . . .                     | 66        |
| 3.5      | Related Work . . . . .                                 | 68        |
| 3.5.1    | Turn Taking Simulations . . . . .                      | 68        |
| 3.5.2    | Thorissón . . . . .                                    | 68        |
| 3.6      | Summary . . . . .                                      | 70        |
| <b>4</b> | <b>Multi-Party Dialogue Management</b>                 | <b>71</b> |
| 4.1      | Introduction . . . . .                                 | 71        |
| 4.2      | Background . . . . .                                   | 72        |
| 4.2.1    | Goffman . . . . .                                      | 72        |
| 4.2.2    | Clark & Carlson . . . . .                              | 72        |
| 4.2.3    | Traum & Rickel . . . . .                               | 74        |
| 4.2.4    | Multilogue QUD . . . . .                               | 75        |
|          | Overhearers . . . . .                                  | 75        |
|          | Side Participants . . . . .                            | 76        |
|          | Duplicate Responders . . . . .                         | 78        |
| 4.3      | Preliminary Discussion . . . . .                       | 80        |
| 4.3.1    | Some words on Utterance Processing Protocols . . . . . | 80        |

|          |   |            |
|----------|---|------------|
| 4.3.2    | Individual, Distributive and Collective Questions . . . . . | 80         |
| 4.3.3    | Challenging the Multilogue Protocols . . . . .              | 82         |
|          | ADR and collective questions . . . . .                      | 82         |
|          | Overhearers and Side Participants . . . . .                 | 84         |
| 4.3.4    | Two Systems of Obligations . . . . .                        | 85         |
| 4.4      | Solving the Problems . . . . .                              | 92         |
| 4.4.1    | AMA protocols . . . . .                                     | 92         |
| 4.4.2    | The Agreement Move . . . . .                                | 94         |
| 4.4.3    | Grounding . . . . .   | 96         |
| 4.4.4    | Obligations . . . . .                                       | 96         |
| 4.4.5    | Extending QUD . . . . .                                     | 97         |
| 4.5      | Summary . . . . .   | 98         |
| <b>5</b> | <b>Implementing a Multi-Party DM</b>                        | <b>99</b>  |
| 5.1      | Modifications of the Turn Manager . . . . .                 | 100        |
| 5.2      | The Information State . . . . .                             | 101        |
| 5.2.1    | Information State Proper . . . . .                          | 102        |
| 5.2.2    | Other Variables . . . . .                                   | 103        |
| 5.2.3    | Plans . . . . .   | 103        |
| 5.3      | The Dialogue Move Engine . . . . .                          | 104        |
| 5.3.1    | The Update Algorithm . . . . .                              | 104        |
| 5.3.2    | The QUD object . . . . .                                    | 105        |
| 5.3.3    | Individual, Collective and Distributive questions . . . . . | 106        |
| 5.3.4    | The Update Rules . . . . .                                  | 106        |
| 5.3.5    | The Selection “Module” . . . . .                            | 112        |
| 5.4      | Integrating Turn and Dialogue management . . . . .          | 115        |
| 5.4.1    | Assigning the Right to Address an Issue . . . . .           | 115        |
| 5.4.2    | Delegating Selection Control to Turn Manager . . . . .      | 116        |
|          | Selection Control in IBiS1 . . . . .                        | 116        |
|          | Selection Control in Multi-IBiS . . . . .                   | 117        |
| 5.4.3    | Unused Features of the Turn Manager . . . . .               | 118        |
| 5.5      | Tracing Multi-IBiS . . . . .                                | 118        |
| 5.5.1    | Agreement and Disagreement . . . . .                        | 118        |
| 5.5.2    | Acceptance and Rejection . . . . .                          | 120        |
| 5.5.3    | Question Assignment . . . . .                               | 120        |
| 5.5.4    | Information Sharing by Overhearing . . . . .                | 121        |
| 5.6      | Summary . . . . .   | 122        |
| <b>6</b> | <b>Conclusions and Future Research</b>                      | <b>125</b> |
| 6.1      | Summary . . . . .   | 125        |
| 6.2      | Concluding discussion . . . . .                             | 126        |
| 6.2.1    | Dialogue Management . . . . .                               | 126        |
| 6.2.2    | SCXML . . . . .   | 127        |

|          |   |            |
|----------|---|------------|
| 6.2.3    | Corpus Studies . . . . .                    | 129        |
| 6.2.4    | QUD and ISSUES . . . . .                    | 130        |
| 6.2.5    | Semantics . . . . .                         | 130        |
| 6.2.6    | Individual Commitments . . . . .            | 130        |
| <b>A</b> | <b>Trace of IBiS1 run</b>                   | <b>137</b> |
| <b>B</b> | <b>Trace of MultiIBiS run: Agreement</b>    | <b>141</b> |
| <b>C</b> | <b>Trace of MultiIBiS run: Disagreement</b> | <b>145</b> |
| <b>D</b> | <b>Trace of MultiIBiS run: Addressing</b>   | <b>149</b> |
| <b>E</b> | <b>Trace of Discussion on Destination</b>   | <b>153</b> |
| <b>F</b> | <b>Trace of Hotel and Price Discussion</b>  | <b>157</b> |

# List of Figures

|     |  |     |
|-----|--|-----|
| 1.1 | Simple dialogue manager . . . . .  | 3   |
| 1.2 | Simple dialogue manager 2 . . . . .  | 4   |
| 1.3 | Parallel dialogue managers . . . . .   | 5   |
| 1.4 | Silence Detection . . . . .  | 9   |
| 1.5 | Multi-party enabled time dialogue manager . . . . .  | 10  |
| 1.6 | Multi-party enabled time agent. Dialogue manager and silence detector run in parallel. . . . . | 11  |
| 1.7 | Multi-party enabled time dialogue manager, refined. . . . .                                    | 13  |
| 2.1 | Simple Statechart . . . . .  | 16  |
| 2.2 | The TrindiKit rule integrateUsrAsk . . . . .   | 24  |
| 2.3 | Update statechart . . . . .  | 32  |
| 2.4 | The Dialogue Move Engine . . . . .   | 34  |
| 2.5 | Parallel control . . . . .   | 36  |
| 2.6 | The IBiS1 statechart . . . . .   | 41  |
| 3.1 | The Overlap State Chart . . . . .  | 55  |
| 3.2 | The TRP State Chart . . . . .  | 58  |
| 3.3 | The Outside Statechart . . . . .   | 63  |
| 3.4 | The Inside Statechart . . . . .  | 64  |
| 3.5 | The improved TRP Chart . . . . .   | 67  |
| 5.1 | Multi-IBiS statechart . . . . .  | 99  |
| 5.2 | The TRP Chart extended with the shared TRP . . . . .   | 101 |
| 5.3 | The Update State of MultiIBiS . . . . .  | 105 |



# List of Tables

|     |   |     |
|-----|---|-----|
| 2.1 | From ISU into Extended SCXML . . . . .  | 26  |
| 2.2 | From TrindiKit into Extended SCXML . . . . .                                    | 37  |
| 3.1 | Events expected as input to the Turn Manager. . . . .                           | 53  |
| 3.2 | Output Events and Output States in the turn manager . . . . .                   | 54  |
| 3.3 | Output expected from the turn manager . . . . .                                 | 60  |
| 4.1 | The roles in spoken interaction according to (Clark and Carlson, 1982). . . . . | 73  |
| 4.2 | Rights and obligations in connection to conversation roles. . . . .             | 91  |
| 5.1 | The fields of the QUD object. . . . .   | 106 |



# Chapter 1

## Introduction

### 1.1 Purpose

The aim of this work is to build artificial conversational agents capable of working in a multi-party setting. By an artificial conversational agent we mean a software agent (possibly with embodiment, personality etc.), capable of engaging in conversation with other conversational agents (human or artificial). By a multi-party setting we mean that more than two agents are engaged in the communication or dialogue.

We imagine an environment where a user interacts with a *collective* of conversational agents, some of which are artificial, some of which are human, perhaps in games (where the agents are in-game characters or player characters), perhaps in an information service (where every agent acts as a representative for one company, institution or domain). Such agents can collaborate and compete with each other, using natural language as their communication protocol.

And exactly that is the purpose of this thesis. We are going to equip the conversational agents with the ability to use natural language, not only as a user interface, but as their general interface for communication.

One reason for avoiding a specialised inter-agent communication protocol is the plug-and-play aspect – as long as an agent has appropriate linguistic coverage of its domain combined with some basic linguistic-pragmatic (for instance turn taking) skills it can be plugged into the agent community. Another reason is increased transparency – a user in this setting can, for example, monitor the inter-agent communication and interrupt it or provide additional information if required.

An agent in a two-participant setup can (almost) safely assume that all user utterances are directed to it, and when the user has finished speaking (or released a push-to-talk button), the floor is free for the agent to make any contribution to the dialogue that it wants (see for instance (Cassell et al., 2000) for a state-based two-party turn management model). In contrast, an agent in a multi-party setting cannot assume that all user utterances are directed to it, since there are other dialogue participants present (Traum, 2003a). Still, it is assumed by the other parties that the agent tries to follow the conversation. Other challenges that multi-party systems face include parallel conversations and assignment of questions to certain Dialogue Participants (henceforth *DPs*).

Since the inter-agent communication turns into a natural language conversation, in which users can participate, it is important that the artificial agents handle turn and dialogue management analogous to the human agents, or in a way that is at least compatible with human dialogue management.

The issues mentioned above touch on two domains of dialogue management research – Turn Management and Sequencing. Turn Management is concerned with who has the right and obligation to speak at what time and for how long. Sequencing deals with the organisation of the dialogue activity – in adjacency pairs (question – answer), repair sequences etc. These domains of dialogue management will be the subject of this thesis.

## 1.2 A simple Dialogue Manager

Let us build ourselves a dialogue manager, the simplest dialogue manager imaginable. For the construction we will use Statecharts (Harel, 1987). In its simplest form, a statechart is just a finite state machine, where transitions from one state to another<sup>1</sup> are triggered by events<sup>2</sup> appearing in a global event queue. Just like ordinary finite-state machines, statecharts have a graphical notation, which we will use to illustrate our examples.

Consider a dialogue system which is capable of speaking about time – or more precisely which can answer simple questions about time, covering (for instance) the dialogue in example 1.1.

```
User: what time is it?  
System: the time is five thirty
```

Example 1.1: A simple dialogue

---

<sup>1</sup>This is a simplification. Transitions in statecharts offers a wealth of possibilities, as we will see in subsequent chapters.

<sup>2</sup>This is also a simplification. Conditions for transitions are very flexible.

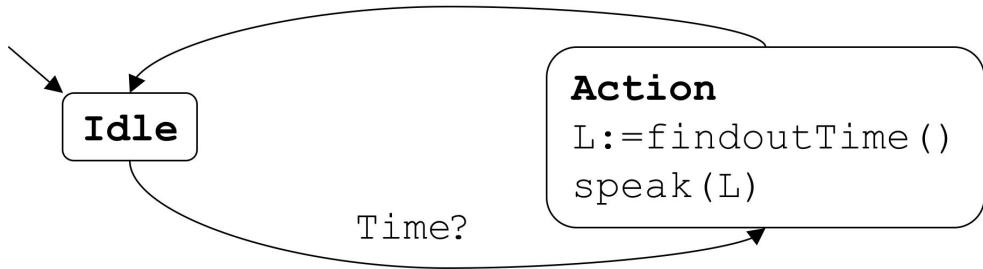


Figure 1.1: Simple dialogue manager

What are the requirements on such a dialogue system? The system should be able to listen for requests for time information, look up the current time and then construct an utterance in which it states the time. Finally the utterance should be produced.

Such a dialogue manager is depicted in figure 1.1. Now, there are a few things to explain in this statechart. The two boxes marked **Idle** and **Action** represent states. The arrows between the states are transitions, similar to the standard graphical representation of finite-state automata.

We do not care about how speech (or text) understanding is done, we just count on receiving events about what has been understood. The transition with source state **Idle** and with the target state **Action** has a label. The label (**Time?**) specifies that an event of the type **Time?** must occur in order for the transition to be carried out. In this state chart we use the event for signalling that a “time request” has been made but we do not care about exactly *what* request for time has been recognised. The utterances “what time is it?”, “do you have the time?”, “do you have a watch?” are possible instances of the **Time?** event. We could have stated this using some formal notation (lambda expressions etc.), but the semantics used in this thesis is so simple that we will do without them.

There are two lines of pseudocode in the **Action** node, which are supposed to be run as soon as we enter the node. The pseudo code calls the function **findoutTime**, and stores the result in variable **L**. The procedure **speak** then produces an output conveying the time expression stored in **L**. We do not care about how this procedure works, if the expression is displayed as text on a screen, uttered by a text-to-speech program etc. as long as the message is communicated to the user.

The transition back to the **Idle** state does not have a label, no event is specified, which means that, as soon as the actions in **Action** have been carried out, we will unconditionally return to the **Idle** state (an  $\epsilon$ -transition in finite state technology terms).

This really simple dialogue manager actually does the job. The state machine starts off in its **Idle** state, and when the user asks “What time is it”, the state machine receives

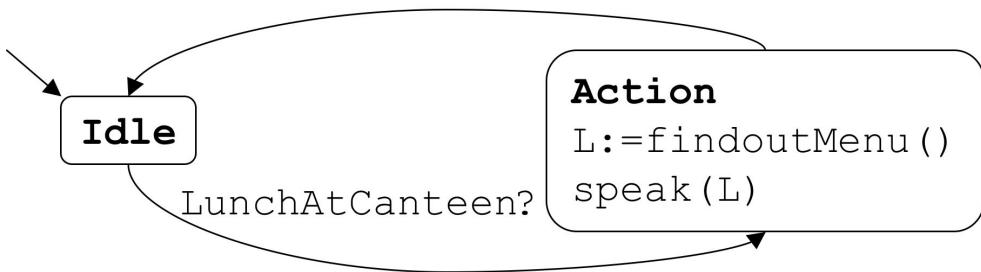


Figure 1.2: Simple dialogue manager 2

a `Time?` event and moves to the `Action` state. Upon entering that state it executes the code contained, communicating to the user the current time, and then moves back into the `Idle` state. Of course, the kind of dialogues that the system handles are not particularly interesting.

## 1.3 From 2-party to N-party dialogue

### 1.3.1 First attempt

Let us introduce another DP (Dialogue Participant) in the dialogue, a third party, also in the form of a dialogue system, or conversational agent, or simply agent, as we will call them from now on.

This new agent does not know anything about time<sup>3</sup>, but on the other hand it does know what they are serving in the university canteen today. The agent's dialogue manager is shown in figure 1.2. The dialogue manager works exactly as the one we saw before. It recognises some (or any) expression, meaning roughly “What are they serving for lunch at the university canteen today?”, and answers it with some utterance stating today’s offering from the university canteen.

The dialogue managers are depicted in Figure 1.3. They are shown inside a box, and the dialogue managers are separated by a dotted line. This is a parallel statechart. The dotted line means that the states that are on each side of the line are both active. They are sending and receiving events to and from the same queue and they can therefore communicate with each other. We will not exploit that fact now – at this point we only want to illustrate the fact that the dialogue managers are both active, listening to the same input events.

---

<sup>3</sup>Maybe it is fair to say that it knows something about time, but not in a way that is interesting for us.

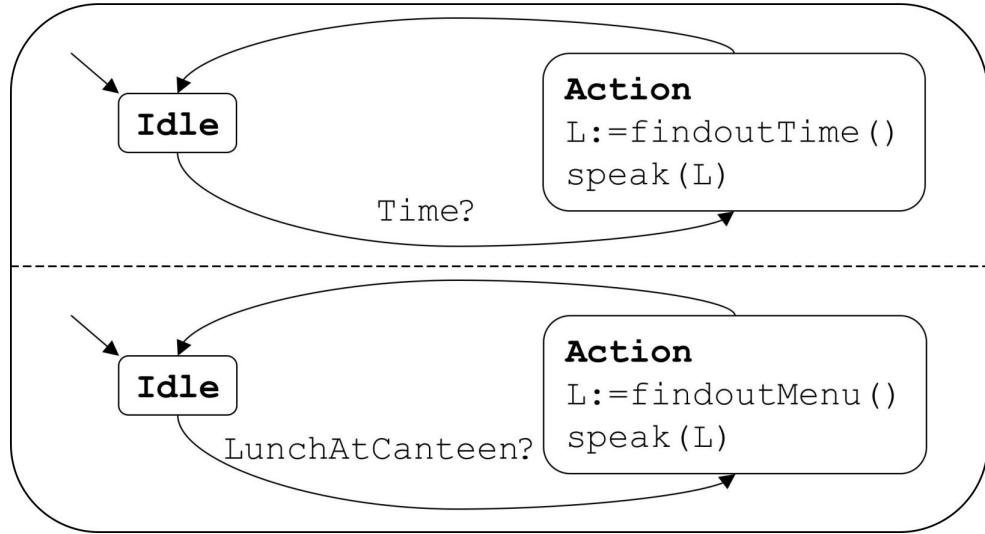


Figure 1.3: Parallel dialogue managers

Let us imagine a short dialogue between a human agent and the two agents. Such a dialogue can be found in example 1.2. The user wants to know the time, and asks, addressing

```

User:      what time is it?
Time Agent: the time is eleven thirty
User:      what do they serve at the university canteen today
Lunch Agent: meatballs

```

Example 1.2: Dialogue with two dialogue systems

the two agents as a collective, a question regarding time. The time agent recognises the question as a time question, and the dialogue manager leaves the initial state **Idle** and enters the **Action** state where it constructs the utterance “the time is eleven thirty” and utters it. It then follows the unlabelled transition back to the **Idle** state. Meanwhile, the lunch agent remains in its **Idle** state, because no transition from that state is labelled with the **Time?** expression. The user gets the time information and thinks that it will soon be time for lunch. He then asks a question about what the canteen serves today. This time the time agent remains in **Idle**, because the event generated by the user request, “**LunchAtCanteen?**” does not allow any transition from that state. The lunch agent, on the other hand, has a transition labelled with that event. The lunch agent then makes a transition from the **Idle** state to the **Action** state, following the arrow labelled **LunchAtCanteen?**. It then constructs the utterance “Meatballs” and utters it. Finally it follows the unlabelled transition back to the **Idle** state.

The user and the agents are taking turns, asking and responding to questions, even though the agents lack explicit turn management abilities. Instead the turn management is implicit

in the agent's dialogue managers. Asking a question about time means that the time-competent agent takes the turn, and asking about lunch at the canteen means that the lunch agent takes the turn. As we shall see in Chapter 3, this way of distributing the turn happens in human-human dialogue, but only as a special case of a more general turn distribution model. If all the DPs are aware of the knowledge distribution among the participants, one could say that the turn is assigned to the different agents by the speaker.

### 1.3.2 Some problematic cases

**Case 1** Imagine a dialogue between a user and two agents equipped with (individual copies of) the dialogue manager depicted in figure 1.1. Let us see what happens if we run these two agents in parallel, asking them a question about what time it is. If we use score notation of the dialogue, it would look like example 1.3.

```
User: what time is it?
Agent 1:           the time is five thirty
Agent 2:           the time is five thirty
```

Example 1.3: Simple dialogue with two dialogue systems (score notation).

There is a problem with the behaviour of the dialogue managers – both agents answer the same question simultaneously. This may not always be a problem. If the DPs speak in chorus as in the example above, the user may still hear what was to be communicated. We can make the problem more obvious by changing the generation mechanism of Agent 2, so that its output is no longer identical to the one of Agent 1. Such a dialogue can be seen in example 1.4.

```
User: what time is it?
Agent 1:           the time is five thirty
Agent 2:           its five thirty
```

Example 1.4: A simple dialogue on time with two time dialogue systems using score notation

The problem is now clearer, since it will probably be impossible to hear what the individual dialogue systems are saying. Similar problems will arise if the agents are not fully co-ordinated in terms of speech rate, choice of words (or syntactic structure), timing of onset of speech etc. If this particular dialogue would occur in a conversation between human DPs it would most certainly be followed by some kind of request for repair and then some kind of repair. Our dialogue managers lack such mechanisms, and this behaviour is thus erroneous.

**Case 2** Imagine a situation where there are two human agents and one time agent equipped with the dialogue manager from figure 1.1. The first human agent asks what time it is, and the second one answers it. The dialogue is shown in the example 1.5.

Human 1: what time is it?

Human 1: its around half past five, I guess

Agent 2: the time is five twenty-eight

Example 1.5: Simple dialogue with one artificial and two human agents, score notation

The second human agent answers the question right away. The artificial agent also answers the question, but in a slightly different way. The result will probably be that the first human agent doesn't catch what any of the responding agents said.

**Case 3** In the third case we have two artificial agents and one human.  $A1$  is an agenda agent (keeping track of a user's meetings etc.),  $A2$  is a time agent and the human agent  $H$  is engaged in dialogue with the two artificial agents. The dialogue is shown in example 1.6, using score notation.

H: where is the meeting

A1: in G312

H: what time

A1: 10:30 a.m. tomorrow

A2: the time is 5:30 pm

Example 1.6: Simple dialogue with one human and two artificial agents with overlapping vocabularies, score notation

$H$  has initiated a dialogue with the agenda agent over a meeting. When  $H$  asks the question "what time", he is probably interested in knowing at what time the meeting starts (although we cannot be certain that he is not interested in the current time).  $A1$  directly answers the question, providing the information regarding the starting time of the meeting. It does so because the utterance of  $H$  is a perfectly valid question for obtaining more information about the meeting that was the topic for the last question.  $A2$  believes that the question is directed to it, because the utterance of  $H$  could be interpreted as a question regarding the current time. In the same way as in the other cases, the agents are speaking at the same time, making their respective utterances hard to hear.

The cases we have described above serve as examples of the constraints that we need to impose on the conversation and on the agents in order for our simplistic, implicit style of turn management to work out. The constraints are described below.

**Human agents must allow artificial agents to answer questions first** If human agents don't give precedence to artificial agents, the human agent and the artificial agent will respond simultaneously – resulting in understanding and perception difficulties.

**The agents' languages must be non-overlapping** As soon as two agents can interpret an utterance, they will try to respond to it. Otherwise, the result will be understanding and perception difficulties.

**The agents' speech and language understanding must be perfect** If the agents' speech and language understanding are not perfect, the risk is that an agent misunderstands something directed to another agent as directed to it and the agents start speaking simultaneously – again resulting in understanding and perception difficulties.

The constraints concerns both the design of the agents and the way a user interacts with the agents. If all of them are imposed on a conversation, the dialogue itself and the usefulness of it will suffer. Constantly allowing the artificial agents to answer the questions first is unnatural (unless, maybe, the human DPs considers the artificial agents to have high social status). If the agents' languages are non-overlapping, the agents cannot interact. Perfect speech and language understanding is the utopia of computational linguistics – it will probably never happen. Let us try to relax the constraints on conversation, by trying some obvious solutions.

### 1.3.3 Some solutions

Let us try to solve the problems by introducing, and making use of, a mechanism for detecting silence, or the absence of speech. Such a mechanism is shown in figure 1.4. It is a simple statechart that keeps track of how many agents (except the owner of the automaton) are speaking at a certain point in time.

There is one state, `SilenceDetector`, encapsulating other states. This kind of state hierarchies are possible in statecharts. A requirement on such complex states is that exactly one substate must be active when the superstate is active. The arrows pointing at the substates indicate that they are the initial states, meaning that as soon as `SilenceDetector` is entered, also `SorS` and then `Silence` become active.

Under the name of the superstate, there is a piece of text, `OnEntry: N := 0`, meaning that as soon as `SilenceDetector` is entered, a variable `N` (the number of speakers) should be assigned the value 0.

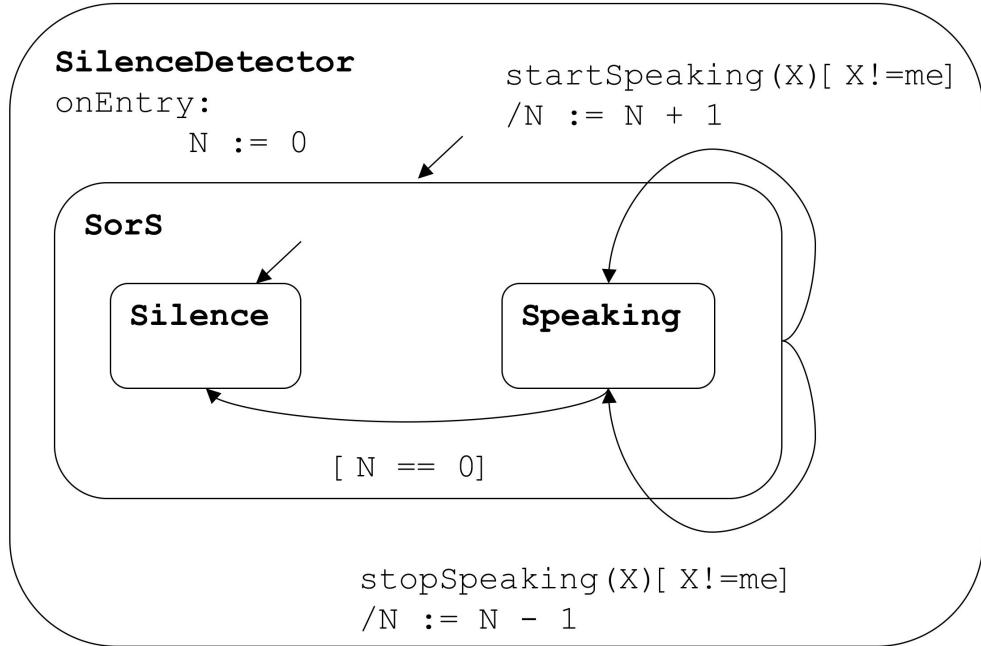


Figure 1.4: Silence Detection

There are two transitions from the state **SorS** to the state **Speaking**. The first one is labelled **startSpeaking(X) [X!= me]** / **N := N + 1**. **startSpeaking** is the name of the event that we want to recognise. This statechart assumes that there is a device somewhere translating the physical event that someone starts speaking into the statechart event with that name. The **X** is a variable containing an identifier of the agent who just started speaking. We use the Prolog/Oz convention – a capitalised name is a variable, a name with a lower-case first letter is a constant (atom). The part within square brackets, **X != me** is a condition, that needs to be fulfilled for the transition to take place. This condition states that the agent who just started speaking should be some other agent than this agent. Finally, after the slash, there is a statement which is executed as the transition is carried out.

The transition means that as soon as someone else than this agent starts speaking, a transition will be made from the state **SorS** to the **Speaking** state, and the variable **N** will be incremented. A transition from a superstate (**SorS**) to a substate (**Speaking**) means that whatever substates to **SorS** are active, they will be exited, and **Speaking** will be entered. The other transition from **SorS** is similar, except that it requires an occurrence of the event **stopSpeaking** and that it decrements **N**.

The transition going from **Speaking** to **Silence**, is labelled with a condition only. An event is not required for this transition, only the fulfilment of the condition **N == 0**. This means that as soon as the number of speakers in the conversation are 0 when **Speaking** is active, a transition from **Speaking** to **Silence** will be made.

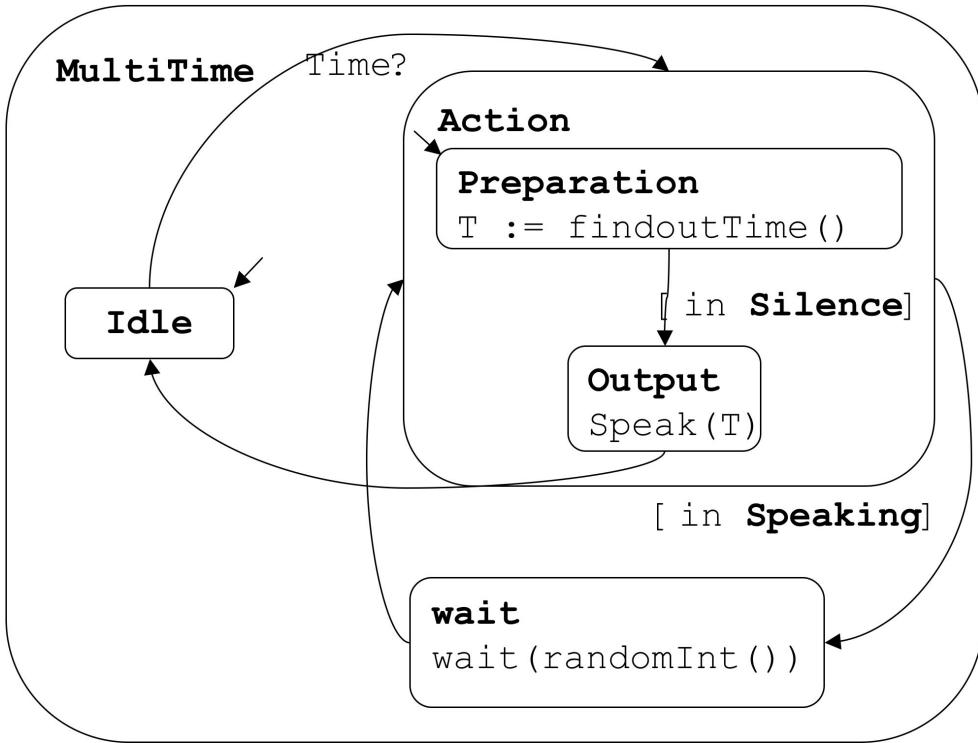


Figure 1.5: Multi-party enabled time dialogue manager

Now that we can detect when other agents are speaking and when they are silent, let us introduce support for this in a dialogue manager. We break the **Action** state of our old time dialogue manager into two sub-states. One of the states becomes a preparation state, assigning the current time to the variable  $T$ . The other state does the actual output of  $T$ . As we have mentioned earlier, using statecharts gives the opportunity to run two states in parallel, reading from and writing to the same queue of events. Another feature, which will be useful to us, is to use information about what parallel states are active as transition conditions.

So, we will run the SilenceDetector in parallel with the new dialogue manager depicted in Figure 1.5. The parallel states can be seen in Figure 1.6. The condition for making the transition between the preparation and output states is that the SilenceDetector is in the state **Silence**, expressed with the condition **[In Silence]**. Another transition has as source the **Action** state and target the **wait** state, and the condition for taking the transition is that the Silence Detector of this particular dialogue manager is in the state **Speaking**.

This dialogue manager is very polite – it is guaranteed to respect other speakers so that it will not interrupt them. Let us see if this dialogue manager can deal with the issues in the examples 1.3, 1.4 and 1.5.

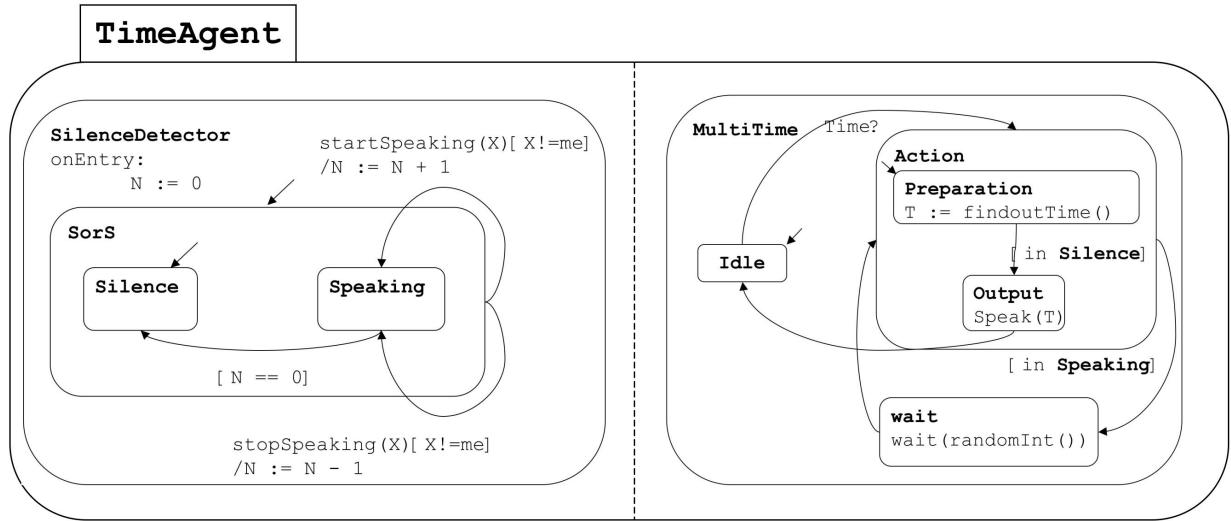


Figure 1.6: Multi-party enabled time agent. Dialogue manager and silence detector run in parallel.

```
(User) what time is it?
(Agent 1)           it ... it's 5:30
(Agent 2)           it ...           it's 5:30
```

Example 1.7: A simple dialogue on time with two time dialogue systems (score notation).

As before, when the user asks the time question, both of the systems recognise the **Time?** event and make the transition to the **Action** state, and then directly to the assignment state, where  $T$  gets the value of a time expression denoting the current time.

When the user stops speaking, the Silence Detector of each dialogue manager will change states from `Speaking` to `Silence`, allowing both agents to speak. As soon as they start speaking, the other agent's Silence Detector will enter the state `Speaking`, thus triggering a transition from `Speak(T)` into the `wait` state, where it will remain for a small random number of milliseconds, before it makes the transition back to `T := findoutTime()`, trying to make a contribution a second time. This time, when both the agents have been randomly delayed, they will not try to speak simultaneously<sup>4</sup>.

Instead Agent 1 will (in this case) make its contribution, and the Silence Detector of Agent 2 will, when hearing the utterance from Agent 1, make the transition from the `Silence` state to the `Speaking` state. This will again trigger the transition to the `wait()` state and then, after the waiting back to the `T := findoutTime()` state. When Agent 1 has finished its contribution, the Silence Detector of Agent 2 will enter the `Silence` state, allowing Agent 2 to make the transition to the `Speak(T)` state, which will finally allow Agent 2 to make its contribution.

So, does this solve our problems? Yes and no. The agents do not try to speak simultaneously any longer, but on the other hand, the agents still do not realize when a question has been answered. By the addition of the transition labelled `Time≈T` from the `Action` state to the `Idle` state, the time agent will not answer the time question if someone else has given a similar answer to the question. The dialogue manager is shown in Figure 1.7. As soon as Agent 2, being in the `Action` state, hears another agent making an utterance with a contents approximately equal to `T`, the transition to state `Idle` will be carried out. The result will be the dialogue shown in Example 1.8.

```
(User) what time is it?
(Agent 1)           it ... it is five thirty
(Agent 2)           it ...
```

Example 1.8: Another simple dialogue on time with two time dialogue systems (score notation).

## 1.4 Discussion

The dialogue managers that we have constructed in this chapter are capable of taking turns (in some sense), they respect other speakers and they do not answer questions that are already answered by other speakers. However, they are not well suited for more complex dialogues as they lack (among other things) the ability to record any form of dialogue

---

<sup>4</sup>Of course, there is a risk that the Agents will use the same random number as a delay. It is also necessary that the delay is not too long.

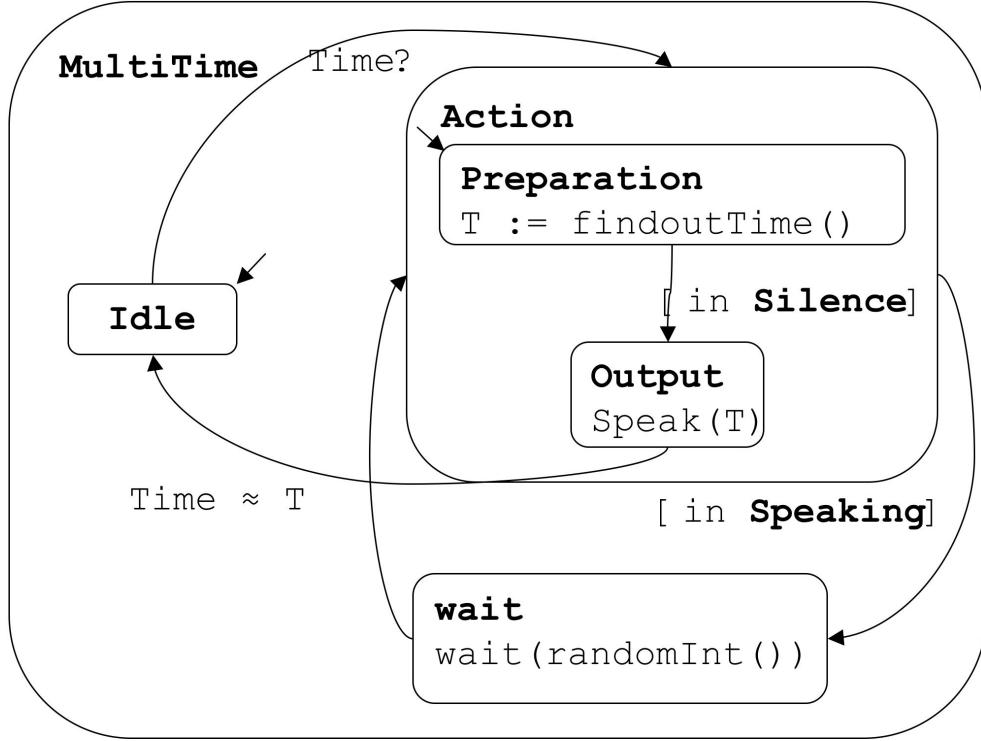


Figure 1.7: Multi-party enabled time dialogue manager, refined.

history, the ability to judge if an utterance is directed to one agent or another, and the ability to do anything other than answering simple questions.

Research on multi-party dialogue management is not common. The main contributor in the area is David Traum, sometimes together with Jeff Rickel. The dialogue manager in (Traum and Rickel, 2002) is a multi-party dialogue manager, but the manager is limited to handle dialogues where there is one single initiator and one single addressee. The layers for managing initiative, turn and grounding all require one single addressee. The same limitation is present in the system described in (Harris et al., 2004), which focus on addressing, turn management and speaker identification, at a relatively practical level.

A number of dialogue management strategies have been suggested and implemented. One of these strategies is the so-called *Issue-based* dialogue management (IBDM) approach (Larsson, 2002), based on Ginzburg's theories about QUD, “Questions Under Discussion”. The theories are originally focused on (human-human) two-party dialogue, but has been extended to cover also multi-party dialogue (“multilogue” in Ginzburg's terms). In Chapter 2 we will describe an implementation of Issue-Based Dialogue Management in SCXML.

In order to find a good turn management system for our agents we will use research on how humans take turns in ordinary conversation. A well-known and established theory of how

humans do turn taking is the one presented by Sacks, Schegloff and Jefferson in (Sacks et al., 1974). We will use that theory as a base for our own turn management system, which is presented in Chapter 3.

In Chapter 4 we will review Fernández and Ginzburg's extensions of the QUD utterance processing protocols and their background in order to lay the theoretical ground for a multi-party issue-based dialogue manager.

In Chapter 5 we will use the findings of Chapter 4 to implement a dialogue manager, integrated with the turn management system described in Chapter 3.

Parts of Chapter 2 has been published as (Kronlid and Lager, 2007). Parts of Chapter 3 has been published as (Kronlid, 2006).

# Chapter 2

## Issue-Based Dialogue Management in SCXML

In this chapter we will explain how the information-state update approach to dialogue management can be implemented in SCXML. We will, after a description of SCXML, start by giving a short introduction to the information-state update model in general, and to issue-based dialogue management in particular. We will present relevant portions of the underlying theory (Ginzburg's DGB and QUD) to IBDM and describe the IBiS family of dialogue managers (Larsson, 2002). We will then use SCXML to implement the IBiS1 system. Along the way, we will also extend SCXML in a manner that makes the implementation of ISU rules easier. Finally, we will show traces of a simulated dialogue with a scxmlIBiS dialogue manager.

### 2.1 SCXML = State Chart XML

SCXML can be described as an attempt to render Harel statecharts (Harel, 1987) in XML. In its simplest form, a statechart is just a finite state machine, where state transitions are triggered by events appearing in a global event queue.

Just like ordinary finite-state machines, statecharts have a graphical notation. Figure 2.1 depicts a very simple example, and other examples can be found in Chapter 1.

Any statechart can be translated into a document written in the linear XML-based syntax of SCXML. For example, the SCXML document capturing the statechart in Figure 2.1 is shown in Listing 2.1. The document can be executed by an SCXML conforming processor,

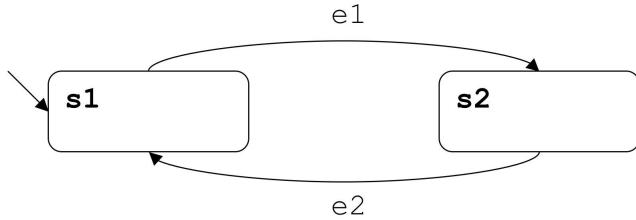


Figure 2.1: Simple Statechart

```

<scxml target="s1">
  <state id="s1">
    <transition event="e1" target="s2"/>
  </state>
  <state id="s2">
    <transition event="e2" target="s1"/>
  </state>
</scxml>
  
```

Listing 2.1: SCXML describing Figure 2.1

greatly simplifying the step from specification into running dialogue system application.

Harel (1987) also introduced a number of (at the time) novel extensions to finite-state machines, which are also present in SCXML, including:

**Hierarchy.** As we have seen in Chapter 1, statecharts may be hierarchical, i.e. a state may contain another statechart down to an arbitrary depth. From a methodological point of view this is important, since it allows us to apply the principles of refinement (a top-down design process in which a state is refined into a number of substates and the transitions between the substates spelled out in detail) and clustering (a bottom-up design process in which a number of similar states are grouped together under the umbrella of a superstate).

**History.** A complex state may contain a history state, serving as a memory of which substate S the complex state was in, the last time it was left for another state. Transition to the history state implies a transition to S.

**Concurrency.** Two or more statecharts may be run in parallel, which basically means that their parent statechart is in two or more states at the same time. This is an important mechanism for introducing independence and orthogonality into a design. We

used this feature in Chapter 1.

**Broadcast communication.** One statechart  $S_1$  may communicate with another statechart  $S_2$  (running in parallel with  $S_1$ ) by placing, in the global event queue, an event that triggers a transition in  $S_2$ . Since the event can in principle be detected by any transition in any state in the statechart, this is often referred to as “broadcast communication”.

**Datamodel.** SCXML gives authors the ability to define a data model as part of an SCXML document. A data model consists of a `<datamodel>` element containing one or more `<data>` elements, each of which may contain an XML description of data.

Closely connected to the datamodel is a *datamodel access language* which also serves as a general scripting language. The value of the `cond` attribute in a `<transition>` element may be a scripting language expression referencing the data, and transitions may thus be conditioned on the data. The `<script>` element and other elements with executable content may be used in actions, or in `<onentry>` or `<onexit>` elements, to modify the data.

The SCXML used in this thesis differs somewhat from the current draft specification (Barnett et al., 2006b). This is due to the fact that the SCXML implementation that we have used during the development of the code used in this thesis is based on an earlier draft (Barnett et al., 2006a). This implementation is the only one available featuring the extension proposed in Section 2.4.2 of this chapter.

## 2.2 The Information State Update Approach to Dialogue Modelling

Simplifying somewhat, a dialogue manager implementing the information state update approach to dialogue modelling can be characterised by the following components:

1. An *information state* representing aspects of common context as well as internal motivating factors
2. A set of *dialogue moves* that will trigger the update of the information state
3. A set of declaratively stated *update rules* governing the updating of the information state

4. An *update strategy* for deciding what rule(s) to select at a given point from the set of applicable ones.

The idea of information state update for dialogue modelling is centered around the information state (IS). Within the IS, the current state of the dialogue is explicitly represented. In Larsson and Traum (2000), the information state of a dialogue is characterised as something that “represents the information necessary to distinguish it from other dialogues, representing the cumulative additions from previous actions in the dialogue, and motivating future action”.

Dialogue moves are meant to serve as an abstraction between the large number of different messages that can be sent (especially in natural language) and the types of updates to be made on the basis of performed utterances (Larsson and Traum, 2000, p. 5). Dialogue moves trigger nonmonotonic updates of the IS. User utterances (or other kinds of user input) are matched against a set of possible update rules that change the IS in the appropriate places (e.g. a new value is entered into a slot). A single user utterance may unleash a whole chain of updates, allowing for generalisations beyond monolithic utterance updates.

The ISU approach should be seen as a rather abstract and relatively “empty” framework that needs to be filled with theoretical content to become a full-fledged theory of dialogue.

Two well-known implementations of the ISU approach to dialogue management are TrindiKit (Larsson and Traum, 2000) and DIPPER (Bos et al., 2003). Implemented/embedded in Prolog and relying to a large extent on properties of its host language, TrindiKit was the first implementation of the ISU approach. In a TrindiKit version, the different modules communicate using so called MIVs (Module Interface Variables) in the information state. The sender writes to the MIV, and the receiver reads from it. To control application of the update rules, algorithms are defined, specifying in what order individual rules or *rule classes* should be tried.

DIPPER is built on top of the Open Agent Architecture (OAA) (Martin et al., 1999), supports many off-the-shelf components useful for spoken dialogue systems, and comes with a dialogue management component that borrows many of the core ideas of the TrindiKit, but is “stripped down to the essentials, uses a revised update language (independent of Prolog), and is more tightly integrated with OAA” (Bos et al., 2003). DIPPER uses OAA “solvables” to interact and communicate with other modules. A solvable is a service provided by another OAA agent. Other implementations exist, but TrindiKit and DIPPER are probably the most important ones.

## 2.3 Issue Based Dialogue Management

### 2.3.1 Ginzburg's DGB and QUD

In his efforts to explain the semantics of questions and answers in dialogue, Jonathan Ginzburg uses the structures *DGB* (DP's Gameboard) and *UNPUB-MS* (Unpublicized Mental Situation) (Ginzburg, 1996). The DGB is a representation of the common ground – what the dialogue participants agree upon (or rather what the DP believes that the participants agree upon). UNPUB-MS is a representation of the mental situation (not known to the other DPs) of a DP.

#### Structure of the common ground

The DGB is an object, quasi-shared among the participants, which is structured as follows: **FACTS** – a set of facts agreed upon by the DPs, **QUD** – questions under discussion – a partially ordered set specifying the currently discussable questions or issues. **LATEST-MOVE** – The contents of the most recent move by any of the DPs. The quasi-sharedness of the DGB means that every DP has its own view of the DGB, but that, if the dialogue is successful, the DGBs of the different DPs are identical or at least very similar.

#### Utterance Processing Protocols

**Querying** The activities in dialogue can be formulated as protocols. The protocol for “cooperative query exchange” looks like this (Ginzburg, 2008):

1. LatestMove.Cont = Ask(A,q): IllocProp
2. A: push q onto QUD; release turn
3. B: push q onto QUD; take turn; make q-specific utterance

The protocol could be expressed as follows: If the latest move is performed by *A* and is a question *q*, then *A* should push *q* on QUD and then release the turn. *B* should also push *q* on QUD and then take the turn and make a *q*-specific utterance. Being “*q*-specific” means that the utterance should be either a proposition *about q* or a question that *influences q*. The formal definitions of aboutness and influence can be found in (Ginzburg, 2008). An informal definition would be something like the following: if *p* gives information that reduces the number of possible answers to *q* (*p* is a partial answer to *q*), it is said to be

*about q*. If the answer to one question  $q_2$  reduces the number of possible answers to another question  $q_1$  ( $q_2$  is a sub-question of  $q_1$ ), then  $q_2$  *influences*  $q_1$ .

**Assertion** The protocol for “cooperative assertion exchange” looks like this (Ginzburg, 2008):

1. LatestMove.Cont = Assert(A,p) : IllocProp
2. A: push p? onto QUD, release turn
3. B: push p? onto QUD, take turn;
  - Option 1: Discuss p?,
  - Option 2: Accept p
  
1. LatestMove.Cont = Accept(B,p) : IllocProp
2. B: increment FACTS with p; pop p? from QUD;
3. A: increment FACTS with p; pop p? from QUD;

So if A has asserted p, both A and B should push the acceptance-question p? on their respective QUD. A should release the turn and B should take the turn. B then has two options. Either accepting p, using an affirmative utterance (“mmh”, “I see” etc.), or start a discussion on p. This discussion can be realised as asking questions or as making assertions. The consequences of an acceptance move is that p should be added to both participant’s FACTS. QUD should then be “downdated” – p should be removed from QUD. Here, the downdating of QUD is a special case of a more general principle: “A question q need to be removed from QUD once FACTS contains an element that resolves q.”

Besides restricting the number of questions open for discussion, QUD has the function of specifying which questions are available for elliptical answers. A question “What did you eat for lunch?” can be answered with the elliptical “pizza” only as long as it is QUD-maximal. Until it is resolved the question will remain on QUD, but it will only be QUD-maximal as long as no more recent (or re-raised) questions will be added to QUD.

### 2.3.2 Larsson’s IBDM

Staffan Larsson develops and implements a theory of Issue-Based Dialogue Management, taking Ginzburg’s concept of QUD as a starting point (Larsson, 2002). QUD (IBiS1 and IBiS2) and the closely related structure ISSUES (IBiS3 and IBiS4) is used to model raising and addressing issues in dialogue (including the resolution of elliptical answers). Issues can also be raised by addressing them, e.g. by giving an answer to a question that has

not been explicitly asked (question accommodation). The implementation is done in four dialogue managers, IBiS1 – IBiS4, each covering different types of dialogue and dialogue phenomena.

Each of Ginzburg’s protocols is broken down into a number of update rules. The group of update rules for integrating dialogue moves are extended with rules for executing plans and for selecting appropriate moves in response to the latest move.

## IBiS1

IBiS1 is introduced with a number of simplifying assumptions, most notably that all utterances are understood and accepted, but also that only a very simplified semantics is required. Another simplification is the removal of the aboutness relation – a relation which is replaced with the *relevance* relation – stating that resolving answers and, in the case of alternative-questions and *wh*-questions, also negated answers are relevant, even if they are not resolving.

Another difference between the accounts of Ginzburg and Larsson is the precise structure of the Information State – the precise structure being a natural requirement since Larsson’s account is formalised as a computer program. Larsson’s Information State is divided into a private and a shared part, corresponding roughly to the UNPUB-MS and the DGB of Ginzburg. The shared part contains **com** – the commitments of the DPs in this conversation (Ginzburg’s FACTS), **qud** – roughly corresponding to Ginzburg’s QUD and **lu** – corresponding roughly to Ginzburg’s LATEST-MOVE. The private part contains the current **plan** – a sequence of actions (in a wide sense) required to gather the information needed to perform an action (for instance answering a certain question), the **agenda** (used for short-term goals) and **bel** (this agent’s beliefs).

Finally, the definitions of QUD differs. In (Ginzburg, 1996, 2008), QUD is a “partially ordered set”, where some questions take precedence over others while others are unordered. As a consequence more than one question can be qud-maximal at a given point. Larsson models QUD as a stack (IBiS1) or an open stack (IBiS2-4) – a stack with set-like properties. In Larsson’s QUD, only one element may be “maximal” – on top of the stack.

The actual logic and update rules of the system will be described in Section 2.4.

## IBiS2

In IBiS2, the simplifying assumption from IBiS1 about perfect communication – that all utterances are understood and accepted – is removed. Update rules for handling grounding (the process of adding to common ground) are added. Ginzburg’s acceptance questions provide a type of grounding, but Larsson argues against a solution using an acceptance question, stating that pushing something on QUD before making the utterance that indicates the QUD update goes against the idea of DGB as a shared structure. Another argument from Larsson is that the explicit acceptance question is only necessary in a dialogue system capable of argumentation, since the acceptance question will be popped immediately after a rejection or acceptance move. It is also noted that while Larsson builds the grounding system in IBiS on four levels (contact, perception, understanding, acceptance), Ginzburg deals only with understanding and acceptance.

Instead, Larsson suggests a solution where the acceptance question is not explicitly represented on QUD. He introduces a field PM (Previous Moves) in the Information State where the moves of the previous utterance are stored. Then, if acceptance feedback occurs, the move which is the subject for the feedback move can be found in the PM field.

Another difference between the accounts of Larsson and Ginzburg is that the former allows rejection and acceptance of issues (questions) as well as of assertions, as in:

A: Where do you want to go  
 B: That’s none of your business

We will not consider acceptance moves in this chapter, but will revisit the issue in Chapter 4.

## IBiS3 and IBiS4

IBiS3 is a dialogue manager capable of handling “question accommodation”, which allows the pushing of a question on QUD even if it has not been explicitly raised. If a move is made, which is an answer to a question that has not been explicitly raised, but could have been, it is allowed to push that question on QUD. To obtain this functionality the structure of QUD is changed – split into two separate structures, ISSUES and QUD. ISSUES contains all the unresolved issues (questions) of the conversation while QUD contains all recently discussed issues, thus available for ellipsis resolution. We will return to the two structures in Chapter 4.

IBiS4 is extended to handle Action Oriented dialogue, and is therefore equipped with an

extra field in the shared part of the Information State, ACTIONS. The field is very similar to ISSUES, but holds a number of “unresolved” actions instead of issues.

## 2.4 Implementing IBiS1 in SCXML

### 2.4.1 A simplified semantics

The semantics of the original IBiS1 system is already very simple (Larsson, 2002). We will go one step further and simplify the already simplified semantics. We will represent a proposition by a combination of a question and an answer. If the question is denoted by  $Q$  and the answer is denoted by  $A$ , the representation of the proposition in our semantics will be  $Q\#A$ . The negation of an answer  $A$  is represented as  $\neg(A)$ , and the negation of a proposition  $Q\#A$  is represented as  $\neg(Q\#A)$ .

A proposition like “I want to go to Paris” is thus represented by the expression `to#paris`. “I don’t want to go to Chelsea” would be represented like `neg(to#chelsea)`.

The semantics is very restricted, but lets us represent simple questions (e.g. “where do you want to go?”, “What class did you have in mind?”, “How can I help you”), answers (like “to Chelsea” (ellipsis), “I want to go to Chelsea”) and negations of propositions and answers. We cannot represent yes/no-questions like “would you like to go to Chelsea?”, alt-questions etc., but for the purposes of this thesis it is all we need.

### 2.4.2 Implementing ISU Dialogue Management

We suggest that most systems implementing the ISU approach to dialogue management can be reimplemented in SCXML. Some of the systems can be translated into standard SCXML, others require a moderate extension for a straightforward implementation.

An information state update rule, formulated in the TrindiKit rule language, allows variables to be bound in the preconditions using a prolog-style condition (at the same time returning a boolean value and binding logical variables). The value of that variable can then be accessed in the effects part of the rule. See for instance the rule “integrateUsrAsk” in Figure 2.2 (Larsson, 2002).

The rule has a name (`integrateUsrAsk`), a set of preconditions (the first prolog list), and a set of effects (the second prolog list). There are two preconditions:

```

RULE: integrateUsrAsk
CLASS: integrate
PRE: { $/SHARED/LU/SPEAKER==usr
      in($/SHARED/LU/MOVES, ask(Q))
EFF: { push(/SHARED/QUD/, Q)
       push(/PRIVATE/AGENDA, respond(Q))

```

Figure 2.2: The TrindiKit rule integrateUsrAsk

1. that the speaker of the latest utterance is **usr** (the user)
2. that one of the moves in the latest utterance was an **ask** move with content **Q**.

The expressions above are evaluating to either true or false. If an expression evaluates to true, the variables in that expression are bound. This means that if a move **ask(time)** is in latest moves, and if the latest speaker was the user, the preconditions will evaluate to true *and* bind the variable **Q** to the value **time**. This value can then be referred to (by using the variable **Q**) in the effects part, where **Q** is pushed on QUD and an action to respond to **Q** is pushed on the agenda.

SCXML supports neither prolog-style queries nor logical variables (as described in the previous paragraph). The current SCXML draft specification (Barnett et al., 2006b) explicitly states that condition checking should have no effects on the datamodel, so the tempting idea to use state-local datamodel entries instead of variables is not possible.

An introduction of variables clearly increases expressivity, but whether that increase is necessary or not is, in our opinion, an open question. The variables are used to store references to certain chunks of data in the datamodel during condition-checking. The references are then used to modify the datamodel in the effects part of the transition. If a certain expression can be used to point out a certain chunk of data in the condition part of the transition, the same expression must point out the same chunk of data in the effects part of the rule, given that the datamodel is not updated between condition-checking and action-application. If the referring expression is used more than once in the effects part, it is necessary to make sure that the datamodel is not updated in such a way that the data needed is modified.

However, the use of variables increases readability and understandability as well as efficiency of the rules since the need to duplicate code is removed. There are more complicated rules than the ones shown in this thesis where large portions of the code for preconditions would need to be duplicated (several times) in both the preconditions and the effects part if variables were not allowed.

SCXML is designed with extensibility in mind (cf Barnett et al. (2006a)), so the solution we

have chosen is to extend the SCXML <transition> element with variables and with prolog-style queries. The logical variables, local to the transition, are declared in an attribute `vars`. The prolog-style queries are probably not necessary, but since our implementation of an SCXML interpreter uses the Oz programming language (which supports this kind of query) as a scripting language, the choice was easy. In another language, without this support, the same behaviour can probably be achieved using (possibly nested) loops (see Listing 2.2 for a sketch of `integrateUsrAsk` in Java). One can argue that the use of Prolog conditions make the translation of the TrindiKit rules easier and more intuitive. We name the result of extending SCXML with the mentioned features *E-SCXML*.

```

private Move m;

public boolean precond() {
    m = null;
    if (IS.shared.lu.speaker == "usr") {
        for (Move i : IS.shared.lu.moves) {
            if (i.getType() == ASK) {
                m = i;
                break;
            }
        }
    }
    return m != null;
}

public void effects() {
    IS.shared.qud.push(m.getContents());
    IS.privat.agenda.push(
        new RespondItem(m.getContents())
    );
}

```

Listing 2.2: A sketch in Java: `integrateUsrAsk`

With this modification to SCXML in place, we believe that most ISU systems can be implemented in E-SCXML, exploiting the mapping between the ISU components and E-SCXML elements depicted in Table 2.1. We cannot really prove this claim, but by taking a simple example system and reimplement it in SCXML we hope to be able to convince the reader of the viability of our approach. We choose to target the IBiS1 system from Larsson (2002), and thus most of our discussion will be comparing TrindiKit with SCXML, but we also hint at how DIPPER compares with SCXML. As we shall see, our conclusion is that SCXML could potentially replace them both.

| The ISU Approach  | (Extended) SCXML |
|-------------------|------------------|
| Information state | Datamodel        |
| Dialogue move     | Event            |
| Update rule       | Transition       |

Table 2.1: From ISU into Extended SCXML

### 2.4.3 Implementing Information States as SCXML datamodels

The expressivity of the SCXML `<datamodel>` is perfectly adequate for representing the required kind of information structures. A typical IBiS1 information state may for example be represented as in Listing 2.3. Here, the datamodel reflects the distinction between what

```

<datamodel>
  <data name="IS">
    <private>
      <agenda>n1l</agenda>
      <plan>n1l</plan>
      <bel>n1l</bel>
    </private>
    <shared>
      <com>n1l</com>
      <qud>[q]</qud>
      <lu>
        <speaker>usr</speaker>
        <move>ask(q)</move>
      </lu>
    </shared>
  </data>
</datamodel>

```

Listing 2.3: The IBiS1 datamodel

is private to the agent that ‘owns’ the information state, and what is shared between the agents engaged in conversation. Note that `IS.shared.qud` points to `[q]`, indicating that it is known by both parties that the question `q` is “under discussion”<sup>1</sup>. `n1l` denotes the empty stack, set, etc..

---

<sup>1</sup>We use `q` and `r` here as placeholders for a question and a response, respectively.

#### 2.4.4 Implementing dialogue moves as SCXML events

The closest SCXML correlate to a dialogue move is the notion of an *event*. An SCXML event has a *name*, and an optional *data payload*. The SCXML draft represents events as XML trees, but for the purpose of this thesis we will represent them as records with a label (for representing their name) and a set of feature-value pairs (for representing the data payload). An ASK move where a speaker *a* is asking a question *q* may thus be represented as: `says(speaker:a move:ask(q))` The considerably bulkier XML representation of the same event would be as follows:

```
<event>
  <name>says </name>
  <data>
    <speaker>a</speaker>
    <move>ask (q)</move>
  </data>
</event>
```

#### 2.4.5 Implementing update rules as transitions

An IS update rule consists of a set of *applicability conditions* and a set of *effects* (Larsson and Traum, 2000, p.5). In SCXML, a distinction is made between conditions on the *name* of the first event in the event queue and conditions on this event's data payload and the datamodel, where the former is the value of the `event` attribute, and the latter the value of the `cond` attribute. Both conditions are optional, meaning that we have the choice of designing transitions which are triggered by event names, by event data, by a particular state of the datamodel, by any combination of the three or by unconditional transitions (see Section 2.4.6).

An effect may be represented by SCXML executable content – the children of the `<transition>` elements. The most frequently used executable content here is `<script>`, for updating the data model. For example, an update rule written as in Listing 2.4 in the Prolog-based TrindiKit notation may look like Listing 2.5 in SCXML. For an explanation of the logic of the rule, see Section 2.4.6.

#### 2.4.6 Implementing the update algorithm as a statechart

Dialogue management involves more than one rule, and the application of the rules needs to be controlled, so that the right rules are tried and applied at the right stage in the

```
rule( integrateSysAsk ,
[ $/shared/lu/speaker = sys ,
$/shared/lu/move = ask(Q)] ,
[ push( /shared/qud , Q ) ] ).
```

Listing 2.4: integrateSysAsk in TrindiKit notation

```
<transition id="integrateSysAsk"
    vars="Q"
    cond="IS.shared.lu.speaker=sys
          IS.shared.lu.move=ask(Q)">
    <script>
        {IS.shared.qud push(Q)}
    </script>
</transition>
```

Listing 2.5: integrateSysAsk in SCXML notation

processing of a dialogue. Furthermore, we require three *kinds* of rules:

1. rules that perform unconditional maintenance operations on the datamodel (representing the information state),
2. rules that enable events (representing dialogue moves) to update the datamodel, and
3. rules that, when triggered by certain states of the datamodel, update it. “Certain states” meaning that some element is present at a certain position, some element is not present etc.

The above example is of the third kind.

In Listing 2.6 we show an example of the first kind of rule, responsible for first clearing the agenda, and then transferring to the `grounding` state. For an example of the second kind of rule we offer the one shown in Listing 2.7. This rule provides a bridge between the events representing dialogue moves and the datamodel representing the information state. If an event of the form `says(speaker:sys move:answer(r))` appears first in the event queue when the statechart is in state `grounding`, the rule will set `IS.shared.lu.move` to the value `answer(r)` and `IS.shared.lu.speaker` to `sys`, and then a transfer to the state `integrate` will take place. The transition differs from the original TrindiKit rule, since the implementation of the move-queue differs. In TrindiKit, the moves are enqueued in the MIV `latest_moves` by the `Interpret` module. In the SCXML implementation, we let the SCXML event queue do the job of the MIV, avoiding duplication of functionality.

```

<state id="init">
  <transition target="grounding">
    <script> {IS.private.agenda clear()} </script>
  </transition>
</state>

```

Listing 2.6: The “init” state

```

<state id="grounding">
  <transition id="getLatestMove"
    event="says"
    target="integrate">
    <assign location="IS.shared.lu.move"
      expr="Eventdata.move"/>
    <assign location="IS.shared.lu.speaker"
      expr="Eventdata.speaker"/>
  </transition>
</state>

```

Listing 2.7: The “grounding” state

In the integration state, shown in Listing 2.8, three transitions representing update rules of the third kind are available. The transition `integrateSysAsk` does the job of integrating an `ask` move performed by the system into the information state. The conditions state that the speaker of the latest utterance should be the system and that the latest move should be an ask move with contents `Q`. If these conditions are fulfilled, the contents of the children of the `<transition>` element is executed. In this case, there is a line of Oz-code stating that the question `Q` should be pushed on QUD. The next transition is `integrateUsrAsk`, which does the job of integrating `ask` moves performed by the user. The difference compared to `integrateSysAsk` is, apart from the expected ones, that the system adds to the agenda an action to respond to the question raised by the user. It is easy to see the correspondence to Ginzburg’s “cooperative query exchange” protocol (see Section 2.3.1).

The last transition is `integrateAnswer`, which integrates answer moves (corresponding to Ginzburg’s assert moves). The conditions to be fulfilled are that the last move should be an `answer` move with contents `R`, that the question `Q` should be QUD-maximal, and that `R` should be a relevant answer to `Q`. The result of applying this rule is to add the proposition that is the result of combining `Q` and `R`. The procedure `Domain.relevantAnswer` corresponds to aboutness/relevance as well as resolvedness in Ginzburg’s and Larsson’s terms. It follows from this simplification that only answers which resolve a question are relevant

```

<state id="integrate">
  <transition id="integrateSysAsk"
    vars="Q"
    cond="IS.shared.lu.speaker=sys
          IS.shared.lu.move=ask(Q)"
    target="downdateQUD">
    <script>
      {IS.shared.qud push(Q)}
    <script>
  </transition>
  <transition id="integrateUsrAsk"
    vars="Q"
    cond="IS.shared.lu.speaker=usr
          IS.shared.lu.move=ask(Q)"
    target="downdateQUD">
    <script>
      {IS.shared.qud push(Q)}
      {IS.private.agenda push(respond(Q))}
    <script>
  </transition>
  <transition id="integrateAnswer"
    vars="Q R P"
    cond="IS.shared.lu.move=answer(R)
          {IS.shared.qud top(Q)}
          {Domain.relevantAnswer Q R}"
          {Lib.combine Q R P}
    target="downdateQUD">
    <script>
      {IS.shared.com add(P)}
    </script>
  </transition>
</state>

```

Listing 2.8: The “integrate” state

to a question<sup>2</sup>. The preconditions make sure that the utterance is relevant with respect to the question on QUD, which is required by Ginzburg’s querying protocol. The action, to add P to the set `IS.shared.com` (Larsson’s version of Ginzburg’s FACTS), implements Ginzburg’s assertion protocol. Given the simplifying assumptions of IBiS1, the parts of that protocol dealing with acceptance can be ignored.

The transitions are tried in document order, and given the current datamodel the last one will be the one chosen for execution. Its effect is that `q#r` (i.e. the proposition which is the result of combining `q` and `r`) will be added to the set at `IS.shared.com`. Thereafter a transition to the state `downdateQUD` (shown in Listing 2.9) will take place.

```

<state id="downdateQUD">
    <transition vars="Q R P"
        cond="{IS.shared.qud top(Q)}
            {Domain.relevantAnswer Q R}
            {Lib.combine Q R P}
            {IS.shared.com member(P)}"
        target="loadPlan">
        <script>
            {IS.shared.qud pop()}
        </script>
    </transition>
    <transition target="loadPlan"/>
</state>
```

Listing 2.9: The “downdateQUD” state

This state implements the “QUD downdate principle”, which (in Larsson’s version) states that “If q is on /shared/qud and there is a proposition p in /shared/com such that p resolves q, remove q from QUD”.

In this state, either the first of its transitions will trigger, first popping the QUD and then leading to the `loadPlan` state, or else the second transition will trigger, also leading to `loadPlan`, but this time without popping the QUD. That is, the state will *try* to downdate the QUD and no error is reported if QUD cannot be downdated. Given the current state of the datamodel in our example, the first rule will trigger, the element on top of the stack at `IS.shared.qud` will be popped, and (the relevant part of) the datamodel will look like Listing 2.10. Note how the underlying FSA ‘backbone’ controls when certain classes of rules are eligible for execution. In statechart notation, the relevant chart can be depicted as in Figure 2.3.

---

<sup>2</sup>This is a simplification which underlines the status of this system as a proof-of-concept. A “real” system for “real” users and “real” applications would of course need a more elaborate view on aboutness and resolvedness.

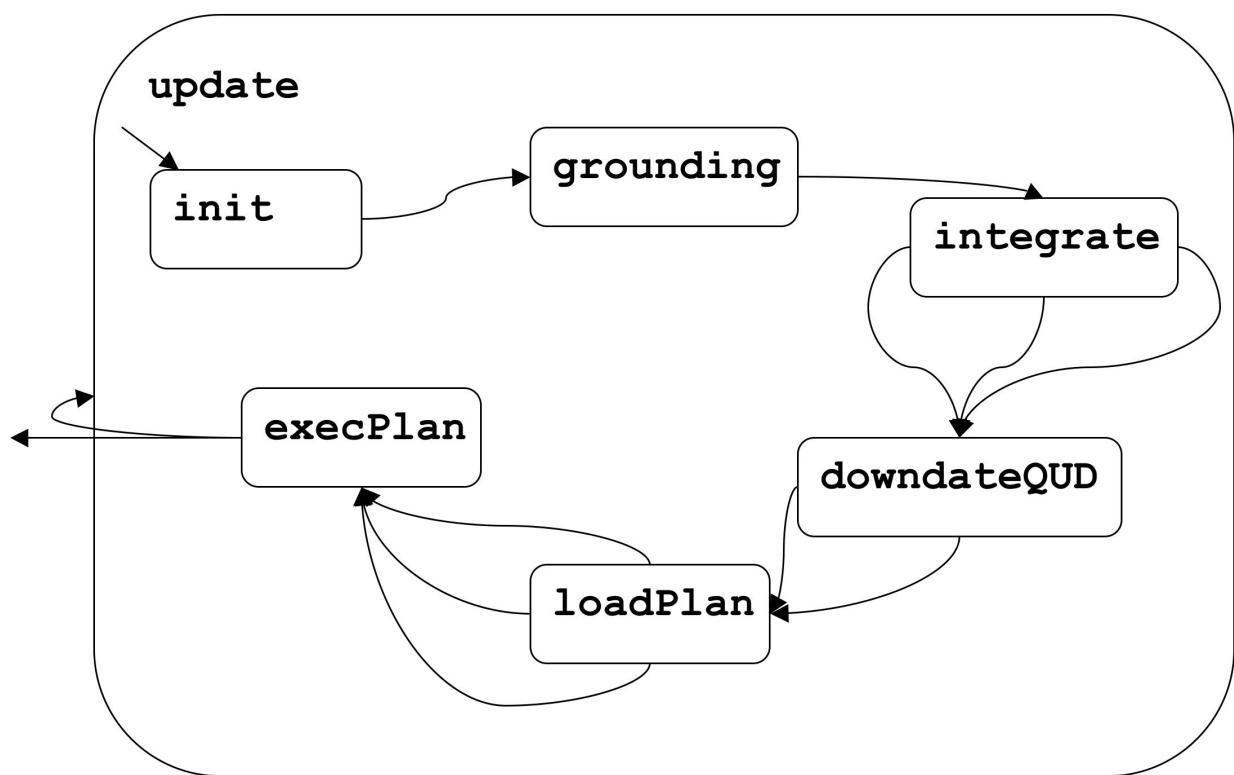


Figure 2.3: Update statechart

```

<datamodel>
  <data name="IS">
    <private>
      <agenda>nil</agenda>
      <plan>nil</plan>
      <bel>nil</bel>
    </private>
    <shared>
      <com>[q#r]</com>
      <qud>nil</qud>
      <lu>
        <speaker>sys</speaker>
        <move>answer(r)</move>
      </lu>
    </shared>
  </data>
</datamodel>

```

Listing 2.10: The datamodel after “downdateQUD” application

By comparison, in TrindiKit the control of the application of update rules is handled by an *update algorithm* written in a procedural language designed for this purpose. The update algorithm (or a version of it) used by IBiS1 is shown in Listing 2.11. Note that

```

if not LATEST_MOVES == failed
then ( init ,
       grounding ,
       integrate ,
       try downdate_qud ,
       try load_plan ,
       repeat exec_plan )

```

Listing 2.11: IBiS1 update algorithm

the statechart in Figure 2.3 does basically the job of this algorithm. Terms like “init”, “grounding”, “integrate”, “downdate\_qud”, etc. refer to TrindiKit rule classes. In our statechart, they correspond to states. To achieve a behaviour corresponding to **try** and **repeat**, we use combinations of targetless and ordinary transitions.

We achieve the *repeat* behaviour by making the transitions (implementing the rules) of a state targetless, so that the state remains active even after a transition has been carried out. The last transition (in document order) is a conditionless transition with a specified target, so that execution can continue in the subsequent state.

The *try* behaviour is implemented by letting all the transitions of a state have a specified target (as in Listing 2.9). The preconditions of each transition are checked, and if a match is found, the transition into the target state is carried out. Not being able to apply a rule is something quite normal, and when this occurs, a conditionless transition (the final one, in document order) is carried out.

### 2.4.7 Implementing the Dialogue Move Engine as a Statechart

The update statechart in Figure 2.3 basically corresponds to the *update module* in IBiS1, responsible for updating the information state based on observed dialogue moves. There is also a *selection module* in IBiS1, responsible for selecting moves to be performed. It does so by translating the actions on the agenda into dialogue moves. In case the agenda is empty, actions are moved from the plan to the agenda.

Together, the update module and the select module forms the *Dialogue Move Engine* (DME) – the dialogue manager proper. As can be seen in Figure 2.4, DME processing starts in the select state and then alternates<sup>3</sup> between update and select.

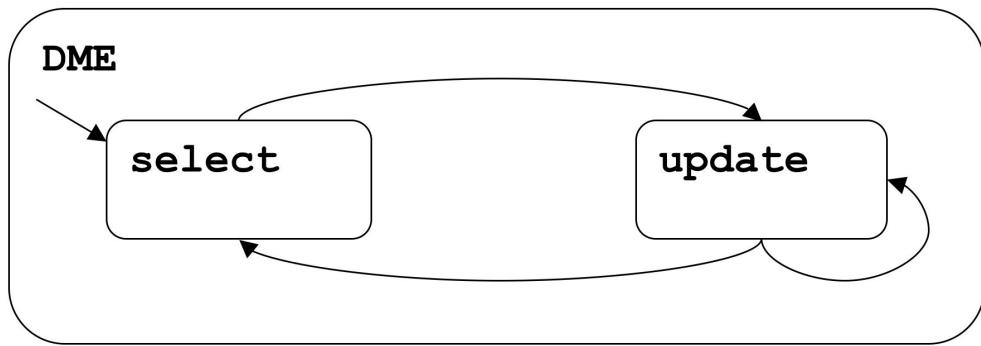


Figure 2.4: The Dialogue Move Engine

A typical dialogue systems does of course contain other modules besides the DME – modules for interpretation and generation for example. They too have their statechart counterparts, as we shall see in the next section.

---

<sup>3</sup>The execution order is determined by the control algorithm, or its SCXML counterpart. The normal execution order is select-update-update.

### 2.4.8 Modules for interpretation and generation

SCXML is not supposed to directly interact with the user. Rather, it requests user interaction by invoking a *presentation component* running in parallel with the SCXML process, and communicating with this component through asynchronous events. Presentation components may support modalities of different kinds, including graphics, voice or gestures. Concentrating on presentation components for spoken language dialogue (a.k.a. “voice widgets”) we may assume that they include things like a Text-To-Speech (TTS) component for presenting the user with spoken information and an Automatic Speech Recognition (ASR) component to collect spoken information from the user.

Such component may of course be very complex, but for all intents and purposes most of their complexity is hidden from the developer, and the only way to communicate with them is through events. The event queues of the invoking and invoked states are separate, and the events of the invoking state reach the invoked state only if they are explicitly directed to it. The invoking state also receives events from the invoked state. The developer must also supply the ASR component with a grammar.

For example, our interpretation module may invoke an ASR component, like shown in Listing 2.12, and our generation module may invoke a TTS component as shown in Listing 2.13.

```
<state id="interpret">
    <invoke target="v3:grammar"
            src="grammar.uxml"/>
</state>}
```

Listing 2.12: ASR invocation

```
<state id="generate">
    <invoke target="v3:audio"/>
</state>
```

Listing 2.13: TTS invocation

### 2.4.9 The dialogue system statechart

The TrindiKit architecture also features a *controller*, wiring together the other modules necessary for assembling a complete dialogue system, either in sequence or through some

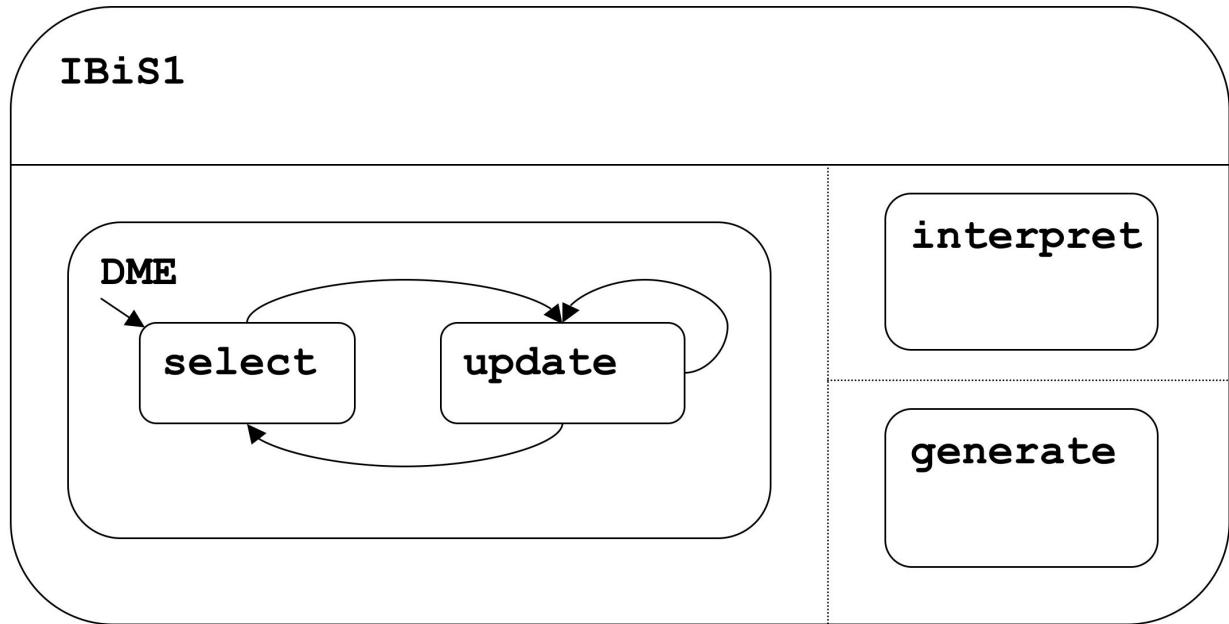


Figure 2.5: Parallel control

asynchronous (i.e. concurrent) mechanism Larsson (2002). We choose here to exemplify an asynchronous architecture, taking advantage of the concurrency offered by SCXML. The statechart corresponding to a full dialogue system might look like in Figure 2.5.

The dotted lines show that the interpretation module, the DME and the generation module are run in parallel. In SCXML the full dialogue system may be sketched as in Listing 2.14

```

<parallel id="IBiS1">
    <state id="interpret" .../>
    <state id="DME" target="select">
        <state id="select" .../>
        <state id="update" .../>
    </state>
    <state id="generate" .../>
</parallel>

```

Listing 2.14: Sketch of full dialogue system

Communication between the modules of the system – between the interpreter, generator and DME – is performed in the broadcast style supported by SCXML, by letting one module place events in the global event queue – events to be picked up by another module.

Comparing SCXML and TrindiKit, we note that the SCXML notion of an event queue

| TrindiKit                  | SCXML           |
|----------------------------|-----------------|
| Information state          | Datamodel       |
| Dialogue move              | Event           |
| Module interface variables | Event queue     |
| Update rule                | Transition      |
| Rule class                 | State (simple)  |
| Update algorithm           | State (complex) |
| Module                     | State (complex) |
| Control algorithm          | State (complex) |

Table 2.2: From TrindiKit into Extended SCXML

seems to do the job of TrindiKit’s module interface variables (MIVs), which is exactly this – to enable modules to interact with each other.

#### 2.4.10 From TrindiKit to SCXML: a summary

In Table 2.2, we summarise the relevant correspondences between TrindiKit and our SCXML formalisation of the ISU approach to dialogue management.

We note that SCXML is considerably simpler than TrindiKit, in that rule classes, update algorithms, modules and control algorithms are all represented as (simple or complex) states/statecharts.

#### 2.4.11 From DIPPER to SCXML

Bos et al. (2003) illustrate the DIPPER architecture and information state update language with an example which implements a “parrot”, where the system simply repeats what the user says. The information state and the relevant update rules, in DIPPER notation, is listed in Listing 2.15. Our SCXML implementation is listed in Listing 2.16. We shall use this example as our point of departure when comparing DIPPER, SCXML and TrindiKit. First, we note that DIPPER uses the *solveables* of OAA for the purpose of enabling modules to interact with each other – for which TrindiKit uses MIVs and SCXML uses the event machinery. In the case of the fourth rule above, a solvable is sent to the OAA agent responsible for speech recognition, which within 10 seconds will bind the variable X to either the recognition result or to the atom `timeout`. This value of X will then be added to the input queue. Our SCXML version works in a similar fashion. An event `recognise`

```

is:record([input:queue(basic),
           listening:basic,
           output:queue(basic)]).

urule(timeout,
      [first(is^input)=timeout],
      [dequeue(is^input)]).

urule(process,
      [non_empty(is^input)],
      [enqueue(is^output,first(is^input)),
       dequeue(is^input)]).

urule(synthesise,
      [non_empty(is^output)],
      [solve(text2speech(first(is^output)),[]),
       dequeue(is^output)]).

urule(recognise,
      [is^listening=no],
      [solve(X,recognise('.Simple',10),
             [enqueue(is^input,X),
              assign(is^listening,no)]),
       assign(is^listening,yes)])).

```

Listing 2.15: DIPPER parrot

is sent in order to activate the speech recognition module, and a transition is triggered by the `recResult` event returned by this module. The `Eventdata` variable will be bound to the recognition result.

Secondly, in the DIPPER rule set, an information state field ‘listening’ is used (as we see it) to simulate a finite state automaton with two states `listening=yes` and `listening=no`. The idea is to control the application of the fourth rule – it is meant to be applicable only in the ‘state’ `listening=no`. The general strategy here appears to be to take advantage of the fact that a production system can easily simulate a finite state automaton. DIPPER can thus eliminate the need for an update algorithm in the style of TrindiKit, but at the expense of complicating the rules.

Note that the ‘listening’ field is not required in the SCXML version, since we can use two “real” states instead. Indeed, looking at TrindiKit, DIPPER and SCXML side by side, we would argue that SCXML provides the neatest and most intuitive solution to the problem of controlling the application of update rules.

```

<scxml initialstate="process">
  <datamodel>
    <data name="IS">
      <input>{New Queue init}</input>
      <output>{New Queue init}</output>
    </data>
  </datamodel>
  <state id="process">
    <transition cond="{IS.input first($)}==timeout">
      <script> {IS.input dequeue} </script>
    </transition>
    <transition cond="{Not {IS.input isEmpty($)}}">
      <script>
        {IS.output enqueue({IS.input first($)})}
        {IS.input dequeue}
      </script>
    </transition>
    <transition cond="{Not {IS.output isEmpty($)}}">
      <send event="speak"
            expr="{IS.output first($$)}"/>
      <script> {IS.output dequeue} </script>
    </transition>
    <transition target="listening"/>
  </state>
  <state id="listening">
    <onentry>
      <send event="recognise"/>
    </onentry>
    <transition event="recResult" target="process">
      <script> {IS.input enqueue(Eventdata)} </script>
    </transition>
  </state>
</scxml>

```

Listing 2.16: SCXML parrot

Finally, few (if any) extensions of SCXML appear to be needed in order to reconstruct DIPPER style dialogue managers in SCXML. This is mainly due to the fact that DIPPER does not make use of Prolog style conditions the way TrindiKit does. Whether the availability of Prolog style conditions in this context is crucial or not is, in our opinion, still an open question. One can argue that it is possible (particularly in the DIPPER case) to write functionally equivalent rules without using prolog conditions, given that variables are bound using stable, deterministic search, and given that the datamodel is not touched between condition checking and action. On the other hand the variables and prolog conditions may simplify the rules, making them more efficient and easier to construct and understand.

## 2.5 Trace of a (scripted) run of SCXML-IBiS1

### 2.5.1 An SCXML Implementation

The example described in this chapter is available at E-SCXML-Lab:

`<http://www.ling.gu.se/~lager/Labs/E-SCXML-Lab/>`.

Visitors are able to try out a number of small examples (including a full version of the example described in this chapter) and are also able to write their own examples, either from scratch, or by modifying the given ones (Kronlid and Lager, 2007).

### 2.5.2 The SCXML Driver

We will use what we call a *driver* state, simulating a user, to test run the SCXML program at the E-SCXML-Lab. The driver is a single state, run in parallel with the rest of IBiS1. The transitions react to the “says” events that the generation state emits, by emitting the appropriate “says”-event in response. For instance, the says event “greet” is responded to by two consecutive events: “greet” and “ask(price)”.

### 2.5.3 The traces

The full trace of a run of IBiS1 can be found in Appendix A. Here we will present some edited, shortened and annotated parts of the trace. The IBiS1 statechart is printed in Figure 2.6.

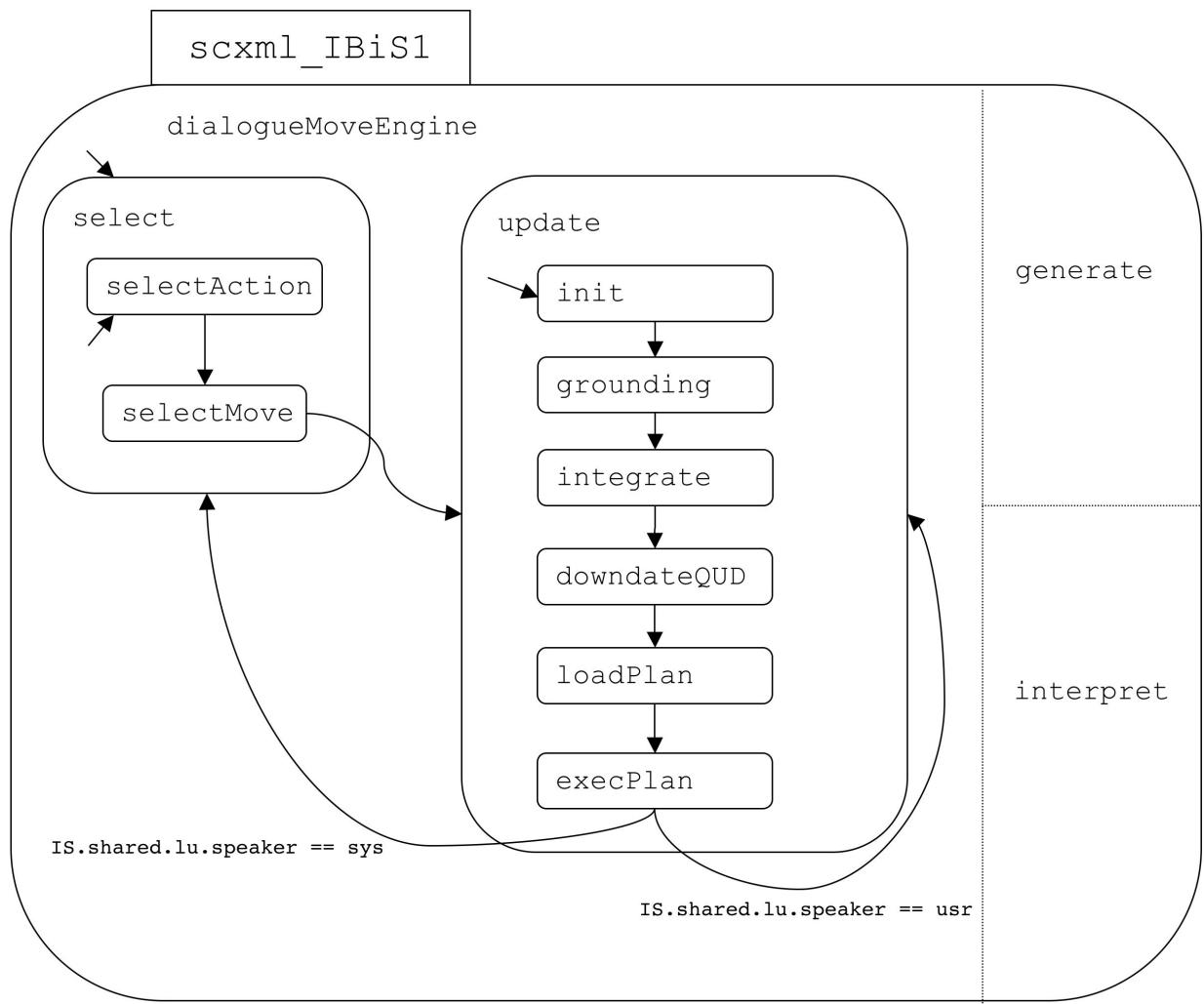


Figure 2.6: The IBiS1 statechart

### Loading a plan to answer a question

At 1580 ms the user utterance is moved into the latest-utterance element of the information state.

|       |      |                     |               |
|-------|------|---------------------|---------------|
| 01470 | Log: | transition applied: | init          |
| 01570 | Log: | transition applied: | getLatestMove |
| 01580 | Log: | usr                 | ask(price)    |

```

<data name="IS">
  <private ... />
  <shared>
    <com .../>
    <qud .../>
    <lu>
      <speaker>usr</speaker>
      <move>ask(price)</move>
    </lu>
  </shared>
</data>

```

At 1680 ms, the asked question is integrated in the information state by means of the `integrateUsrAsk` transition. The `price` question becomes QUD-maximal, and the action to respond to that question is pushed on the agenda.

```

01580    Trans:      grounding:<s> => [says] => integrate:<s>
01680    Log:        transition applied:      integrateUsrAsk
01680    Trans:      integrate:<s> => downdateQUD:<s>

<transition id="integrateUsrAsk" vars="Q" target="downdateQUD"
            pcond="IS.shared.lu.speaker=usr
                  IS.shared.lu.move=ask(Q)">
  <script>
    {IS.shared.qud push(Q)}
    {IS.private.agenda push(respond(Q))}

  </script>
  <log label=""'transition applied:'" expr="integrateUsrAsk"/>
</transition>

```

```

<data name="IS">
  <private>
    <agenda> [respond(price)] </agenda>
    <plan> nil </plan>
    <bel> nil </bel>
  </private>
  <shared>
    <com> nil </com>
    <qud> [price] </qud>
    <lu>
      <speaker>usr</speaker>
      <move>ask(price)</move>
    </lu>
  </shared>
</data>

```

It is not possible to downdate QUD at this stage, so the next interesting transition is `loadPlanT`. The conditions are fulfilled – there is an action to respond to the price question on the agenda, a relevant answer to the price question exists, and we have no beliefs on the issue of price.

The action part of the transitions pops the item on the agenda (since we do not yet have an answer to the question), and loads the plan for finding out the answer to the question. The transition contains hard coded references to the price issue and the price plan, but is easily parameterised (a function for finding plans is required).

```

01690      Trans:      downdateQUD:<s> => loadPlan:<s>
01720      Log:       transition applied:      loadPlanT
01720      Trans:      loadPlan:<s> => execPlan:<s>

<transition id="loadPlanT" vars="P" target="execPlan"
            pcond="{IS.private.agenda top(respond(price))}"
            {Domain.relevantAnswer price P}
            {IS.private.bel nonMember(price#P)}>
  <script>{IS.private.agenda pop()}</script>
  <assign location="IS.private.plan" expr="Plan"/>
</transition>

```

The information state after the application of the `loadPlanT` transition is shown below. The agenda is empty, but the plan contains a list of actions, which if carried out, will provide us with the data needed to consult a database for the response.

After the plan has been loaded, a transition is made to `select`, in order to start the processing of the plan.

```
<data name="IS">
  <private>
    <agenda> nil </agenda>
    <plan> [findout('how') findout('from') findout('to')
             findout('month') findout('class') consultDB(price)
             respond(price)]
    </plan>
    <bel> nil </bel>
  </private>
  <shared>
    <com> nil </com>
    <qud> [price] </qud>
    <lu>
      <speaker>usr</speaker>
      <move>ask(price)</move>
    </lu>
  </shared>
</data>
```

01750      Trans:      execPlan:<s> => select:<s>

### Question – Answer

The first transition is `selectActiont`, which given that the agenda is empty, copies a plan entry to the agenda. In this case the action to ask how the user wants to travel is added to the agenda.

01810      Trans:      selectAction:<s> => selectMove:<s>
01810      Log:        transition applied:      selectAction

```
<transition id="selectActiont" vars="Action" target="selectMove"
           pcond="{IS.private.agenda isEmpty()
                  {IS.private.plan top(Action)}"'}
           <script>{IS.private.agenda push(Action)}</script>
</transition>
```

At 1850, the transition `selectAsk` is applied, since the action to findout the answer to the `how` question is topmost on the agenda.

```

01840    Event:      generateMove      o(move:ask(how))
01850    Log:        transition applied:      selectAsk
01850    Trans:      selectMove:<s> => update:<s>

<transition id="selectAsk" vars="Q" target="update"
            pcond="{IS.private.agenda top(findout(Q))}"
            [] {IS.private.agenda top('raise'(Q))}">
  <send target="Self"
    event="generateMove"
    expr="o(move:ask(Q))"/>
</transition>

```

At 2020 ms, the question about how the user wants to travel is integrated (pushed on QUD). Note that the action `findout(how)` still remains in the plan.

```

01880    Event:      says      o(move:ask(how) speaker:sys)
01960    Event:      says      o(move:answer(train) speaker:usr)
02020    Log:        transition applied:      integrateSysAsk

<transition id="integrateSysAsk" vars="Q" target="downdateQUD"
            pcond="IS.shared.lu.speaker=sys
                  IS.shared.lu.move=ask(Q)">
  <script>{IS.shared.qud push(Q)}</script>
</transition>

```

```

<data name="IS">
  <private>
    <agenda> nil </agenda>
    <plan> [findout(how)
              findout('from')
              findout('to')
              findout(month)
              findout('class')
              consultDB(price)
              respond(price)
            ]
  </plan>
  <bel> nil </bel>
</private>
<shared>
  <com> nil </com>
  <qud> [how price] </qud>
  <lu>
    <speaker>sys</speaker>
    <move>ask(how)</move>
  </lu>
</shared>
</data>

```

At 02370, the user answer to the how question is integrated. The answer R is paired with Q, the topmost QUD element, and added to the commitments.

02370      Log:      transition applied:      integrateAnswer

```

<transition id="integrateAnswer"
  vars="Q R"
  pcond="IS.shared.lu.move=answer(R)
        {IS.shared.qud top(Q)}
        {Domain.relevantAnswer Q R}"
  target="downdateQUD">
  <script>{IS.shared.com add(Q#R)}</script>
</transition>

```

```

<data name="IS">
  <private>
    <agenda> nil </agenda>
    <plan> [findout(how)
              findout('from')
              findout('to')
              findout(month)
              findout('class')
              consultDB(price)
              respond(price)
            ]
  </plan>
  <bel> nil </bel>
</private>
<shared>
  <com> [how#train] </com>
  <qud> [how price] </qud>
  <lu>
    <speaker>usr</speaker>
    <move>answer(train)</move>
  </lu>
</shared>
</data>

```

At 02390, QUD is downdated: the `how` question is removed, since there is a commitment to `how#train`.

```

02390      Log:      transition applied:      downdateQUDt
02390      Trans:     downdateQUDt:<s> => loadPlan:<s>

<transition id="downdateQUDt"
            vars="Q R"
            pcond="{IS.shared.qud top(Q)}
                  {Domain.relevantAnswer Q R}
                  {IS.shared.com member(Q#R)}"
            target="loadPlan">
  <script>{IS.shared.qud pop()}</script>
</transition>

```

At 02480, the `findout(how)` entry of the plan is finally removed, since there is a joint commitment regarding the `how` question.

```

02480      Log:      transition applied:      removeFindout

<transition id="removeFindout"
    vars="Q A"
    pcond="{IS.private.plan top(findout(Q))}"
        {IS.shared.com member(Q#A)}
        {Domain.relevantAnswer Q A}">
    <script>{IS.private.plan pop()}</script>
</transition>

<data name="IS">
    <private>
        <agenda> nil </agenda>
        <plan> [findout('from')
            findout('to')
            findout(month)
            findout('class')
            consultDB(price)
            respond(price)
        ]
        </plan>
        <bel> nil </bel>
    </private>
    <shared>
        <com> [how#train] </com>
        <qud> [price] </qud>
        <lu>
            <speaker>usr</speaker>
            <move>answer(train)</move>
        </lu>
    </shared>
</data>

```

## 2.6 Summary

In this chapter we started by presenting SCXML, the XML rendering of statecharts which is used as implementation language throughout this thesis. We described what characterizes an ISU dialogue manager, and presented Issue Based Dialogue Management, explaining its relation to Ginzburg's theories of QUD and utterance processing protocols, and highlighting a few of the differences between the theories of Larsson and Ginzburg (acceptance questions, QUD etc.).

After a short look at Larsson's various IBiS systems, we implemented IBiS1 in SCXML, taking note of the neat correspondence between the building blocks of ISU systems and SCXML. We finished by showing a trace of a simulated interaction with the SCXML implementation of the dialogue manager.



# **Chapter 3**

## **Turn Management**

### **3.1 The Need for a Turn Management System**

Sacks et al. (1974) describes turn management as an economy. There is something of value – the turn – which is to be allocated and distributed among the participants in a conversation. The turn organisation model that they suggest is supposed to account for a number of “grossly apparent facts” concerning turn-taking (that (as a general rule) only one DP speaks at a time, that speaker change occurs, that turn-order varies etc.). They deliberately do not present the rationale of the facts.

Duncan gives the following explanation of why people take turns speaking (Duncan, 1972, p.298):

Just as it is desirable to avoid bumping into people on the street, it is desirable to avoid in conversations an inordinate amount of simultaneous talking. Beyond considerations of etiquette, it is difficult to maintain adequate mutual comprehensibility when participants in a conversation are talking at the same time.

According to (Allwood, 1995), turn taking is management regarding who may speak, about what topic, at what time and for how long and in what manner. The need for turn-taking arises from the constraints that the human information processing ability imposes. We cannot both send and receive a large number of simultaneous messages. We cannot simultaneously receive a large number of messages from different sources. In order for efficient communication to take place, turns need to be managed. Also ethical reasons are

mentioned – that turn-management is needed in order for all the DPs to have a fair chance to communicate.

The turn manager that we will build will help an agent speaking at the right time. The other issues (topic, size of contributions and speaking manner) will need to be taken care of in other parts of the dialogue system.

## 3.2 Turn Manager Requirements

**Theoretical foundation** The turn manager implementation must be theoretically founded, in order to be compatible with and useful for users of the system. With theoretically founded, we mean that the turn manager must build on research on how humans take turn.

An influential and widely accepted model for how turns are managed in human-human conversation is the so called *SSJ*<sup>1</sup> model (Sacks et al., 1974). The essence of the model is as follows: A Turn Constructional Unit (TCU) is a phrase, clause, sentence or word with a predictable endpoint. A TCU corresponds more or less to an utterance<sup>2</sup>. The first possible completion of a TCU constitutes a Transition Relevance Place (TRP) – a place where speaker-change is possible (or preferred). The occurrence of a TRP can be predicted with relatively high precision by the DPs. The turn transitions are governed by the following two rules:

1. For any turn, at the first TRP of the first TCU
  - (a) The speaker may select the next speaker. In this case, the person selected is the only one with the right and obligation to speak.
  - (b) Else, the next speaker may self-select. The first person to speak acquires the right to a turn.
  - (c) Else, the current speaker may but need not continue.
2. Rules 1 (a–c) apply for each next TRP of this TCU until transfer is effected.

Also, repair mechanisms are included in the model, for instance overlap resolution and clarification of selection problems (for instance, when one DP believes that she is selected as the next speaker when she is not, and vice versa).

---

<sup>1</sup>SSJ because of the first letters in the authors last names - Sacks Schegloff Jefferson

<sup>2</sup>The original definition is more complex, but for our purposes this definition suffices.

**Functionality** The turn manager assumes that there is a device, somewhere in the agent, which signals certain events in the dialogue to the turn manager. These events are shown in table 3.1. Eventdata is represented in parenthesis after the event name.

|                               |                           |
|-------------------------------|---------------------------|
| <code>startSpeaking(X)</code> | speaker X starts speaking |
| <code>stopSpeaking(X)</code>  | speaker X stops speaking  |
| <code>addressing(X)</code>    | speaker X is addressed    |

Table 3.1: Events expected as input to the Turn Manager.

There are a number of states and events in the dialogue that a SSJ turn manager should be able to detect and make visible for the rest of the agent.

1. When this agent is the only DP (Dialogue Participant, or Dialogue Partner) with the right and obligation to speak at this TRP
2. When an agent other than this is the only DP with the right and obligation to speak at this TRP
3. When anyone (including this agent) may self-select at this TRP
4. When this agent's current contribution is overlapping someone else's
5. when the abovementioned overlapping has been resolved.

The list could be longer, including support for more repair mechanisms than overlaps etc., but for a prototype, the list is of the right size.

The SSJ model does allow for brief overlaps, but specifies that when two parties speak simultaneously, this must be resolved. The general SSJ resolution to simultaneous talk is that “first starter goes”, in other words that the DP that starts speaking first at a TRP gets the right to the turn. Interruptions are another type of overlap that also needs to be resolved. In this chapter, when building our turn manager, we will not be concerned with how to resolve such conflicts – we just want to know when there is one and when it has been resolved. The actual resolution is to be carried out by another part of the dialogue system.

There is one thing missing from the list above. Sacks et al. (1974) mentions the possibility of selecting the next speaker in terms of “social identity”. To illustrate this a conversation is used, involving two couples, where a DP from one couple selects a member of the other couple as the next speaker. We will not examine this mechanism in detail until Chapter 4, where it will be discussed in connection with addressing of multiple DPs.

| Name            | Meaning  | Type  |
|-----------------|--|-------|
| myTRP           | This agent is selected as the next speaker                       | state |
| overlap         | this agent's speech is currently overlapping someone else's      | state |
| overlapResolved | this agent's speech is not overlapping someone else's any longer | event |

Table 3.2: Output Events and Output States in the turn manager

### 3.3 A Two-party Turn Manager

Let us start with a simpler problem. We reduce the complexity of the problem by restricting the number of dialogue partners to two. The events we require the Turn Manager to emit and the states that can be inspected with the `In` predicate can be found in table 3.2. Although this TM is just for pedagogical reasons, it may be the case that it is useful for dialogue systems and conversational agents that operate in a two-party setting.

One may consider a model where it is only signalled if this agent has the *right* to speak right now, but then the obligation dimension of the selection mechanism is lost (see Chapter 4 for more issues on obligations in dialogue). We also want to know if a TRP has been cancelled (i.e. someone, this agent or the other party, has started speaking). Another field of interest concerns overlaps. We want to know when this agent's speech is overlapping someone else's and when such an overlap has been resolved.

The Turn Manager will be represented as an SCXML document, and as such a Harel statechart. We will start by sketching the part that deals with overlap. This is maybe not the most important part, but it may be of help to have it defined when we turn to the TRP issues.

#### 3.3.1 Overlaps

A resolved overlap means that one of the speakers has stopped speaking. It *could* mean that both DPs stopped speaking at the same time, but due to the implementation of the event handling (a queue) we may safely assume that in our model two things (two events) may not happen (occur) at exactly the same time. Hence, one of the speakers has stopped speaking. An overlap means that two DPs are speaking at the same time. This, in turn, means that one DP was speaking, and then the other one started speaking. Since we know that there are only two speakers present, we know that one of the speakers is this agent. This description of overlaps leaves us with a rather clear view of what states are needed

to describe overlaps and their resolution.

The first state is `silence`. `silence` means that neither of the DPs are currently speaking. The second state is `oneDpSpeaking`, indicating that exactly one DP is currently speaking. The third state is `overlap`, with the obvious meaning. We add a transition from `silence` to `oneDpSpeaking`, labelling it with the event `startSpeaking`. We need not care of who has started speaking, which is why the Eventdata is omitted. We also add an identically labelled transition from `oneDpSpeaking` to `overlap`. From `overlap` to `oneDpSpeaking` and from `oneDpSpeaking` to `silence` we add transitions labelled with the event `stopSpeaking`.

Now we have a statechart which is capable of tracking if the DPs are silent, if one of the DPs is speaking or if two DPs are speaking at the same time and thus overlapping each other's speech. By using the `In` construction of SCXML, another parallel state can detect if the statechart is in the `overlap` state or not. We just have to make sure that the chart emits the event we are interested in, `overlapResolved`. We add it as an `onExit` action in the `overlap` state (`onExit: overlapResolved`). This means that the event `overlapResolved` will be emitted as soon as we exit the state. The Overlap state chart is depicted in Figure 3.1.

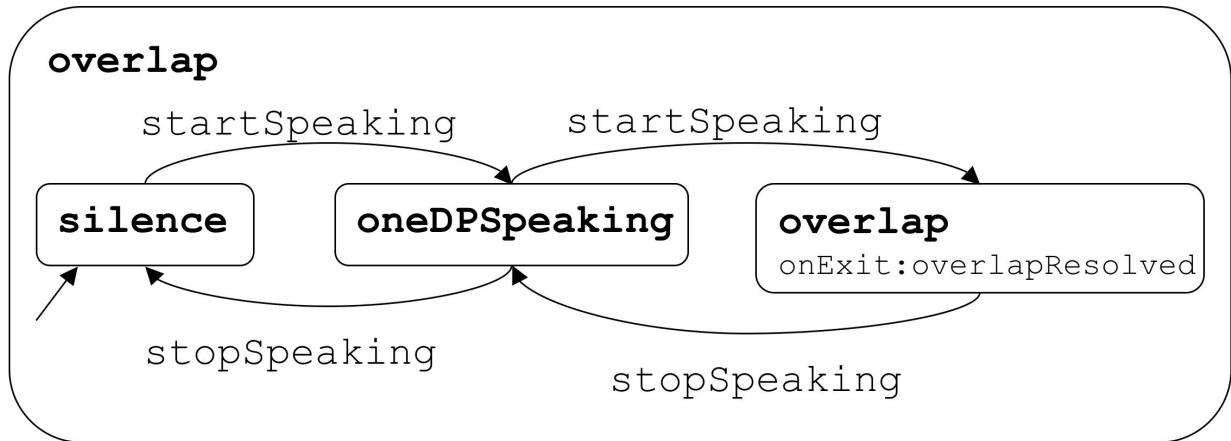


Figure 3.1: The Overlap State Chart

### 3.3.2 TRP management

Let us now turn to the TRPs. We will make use of the possibility to have orthogonal states in a statechart, introducing a state parallel to the `overlap` state. We name the new state, which will manage the indication of TRPs, `trpChart`. The parallel statecharts share the same event queue, and can therefore send and receive events to and from each other.

## Free TRPs

We assume that the machine starts out in a state called `freeTRP`. That is, we assume that in the start of a dialogue, anyone has the right to self-select to make a contribution. There may be dialogues where this is not right, but as we shall see, it is easy to change the initial TRP into another type by simply posting an event to the chart. To test for if there is currently a free TRP, interested parallel statecharts can use the `In` function, as in “`In freeTRP`”. The expression will evaluate to true only if the statechart is currently in the `freeTRP` state.

Unless the current speaker selects the other speaker as the next speaker, the upcoming TRP will be a free TRP. Let us introduce the state which we name `freeTRPComingUp` for the situation when there is currently no TRP and when the next TRP, as far as we know, will be a free TRP. The state will not have any `onEntry` or `onExit` actions, but will only exist to internally keep track of the dialogue state.

As soon as the current speaker stops speaking, we will make a transition to the `freeTRP` state. How should this statechart keep track of who is speaking and who is not? It does not have to, since our overlap machine already does this for us. We simply add a transition from `freeTRPComingUp` to `freeTRP` with the label `[In silence]`, meaning that the transition will be carried out only if the condition that the orthogonal machine is in the state `silence` evaluates to true.

## This agent’s TRPs

We have now covered the event `overlapResolved` and states `freeTRP` and `overlap`. There is now only one of the required states (from Table 3.2) left to take care of – `myTRP`, indicating that this agent is the only DP with the right and obligation to speak. As stated earlier, this occurs when this agent has been selected as the next speaker by the other speaker, and the other speaker has ceased speaking. Selection is signalled with the event `addressing(X)`. We add two states corresponding to the ones we already have – `myTRPComingUp` and `myTRP` – and also a transition from the first into the second with the label `[In silence]`. This means that when this agent is addressed (the state `myTRPComingUp` is active) and silence occurs, the transition into the state `myTRP` is made.

We also need a transition from `freeTRPComingUp` to `myTRPComingUp`, labelled with the event `addressing(X) [X == me]` to make sure that the state `freeTRPComingUp` is active when this agent is selected as the next speaker. The event specification `addressing(X) [X == me]` means that the agent addressed must evaluate to this agent’s name (or identifier).

### Other agents' TRPs

Let us take a look at the case when the other DP is selected as next speaker (i.e. this agent addresses the other DP). If we deploy the TM as it looks right now, the TRP occurring when the other DP is the only one with the right and obligation to speak, will show up as a `freeTRP`, which is not what we desire.

We solve this by adding another state, `otherSelected`, which we want to become active as soon as the other party has been addressed. We achieve this by adding a transition from the state `freeTRPComingUp` to the `otherSelected` state, labelled `addressing(X) [X!=me]`.

### 3.3.3 Making it complete

Our TM is almost ready, but there is an important transition missing. We mentioned in the “Free TRPs” paragraph above that we leave the `freeTRP` state when someone starts speaking, and that we should then enter a state `freeTRPComingUp`, but we did not mention how we should get there.

We need to react and change states whenever someone starts speaking – no matter what state is currently active. In statecharts and SCXML this can be accomplished by wrapping all the TRP states into one superstate and then by adding a transition going from that superstate to `freeTRPComingUp`. We label the transition `startSpeaking`. This means that every time that someone starts speaking, we will end up in the `freeTRPComingUp`. The chart can be seen in figure 3.2.

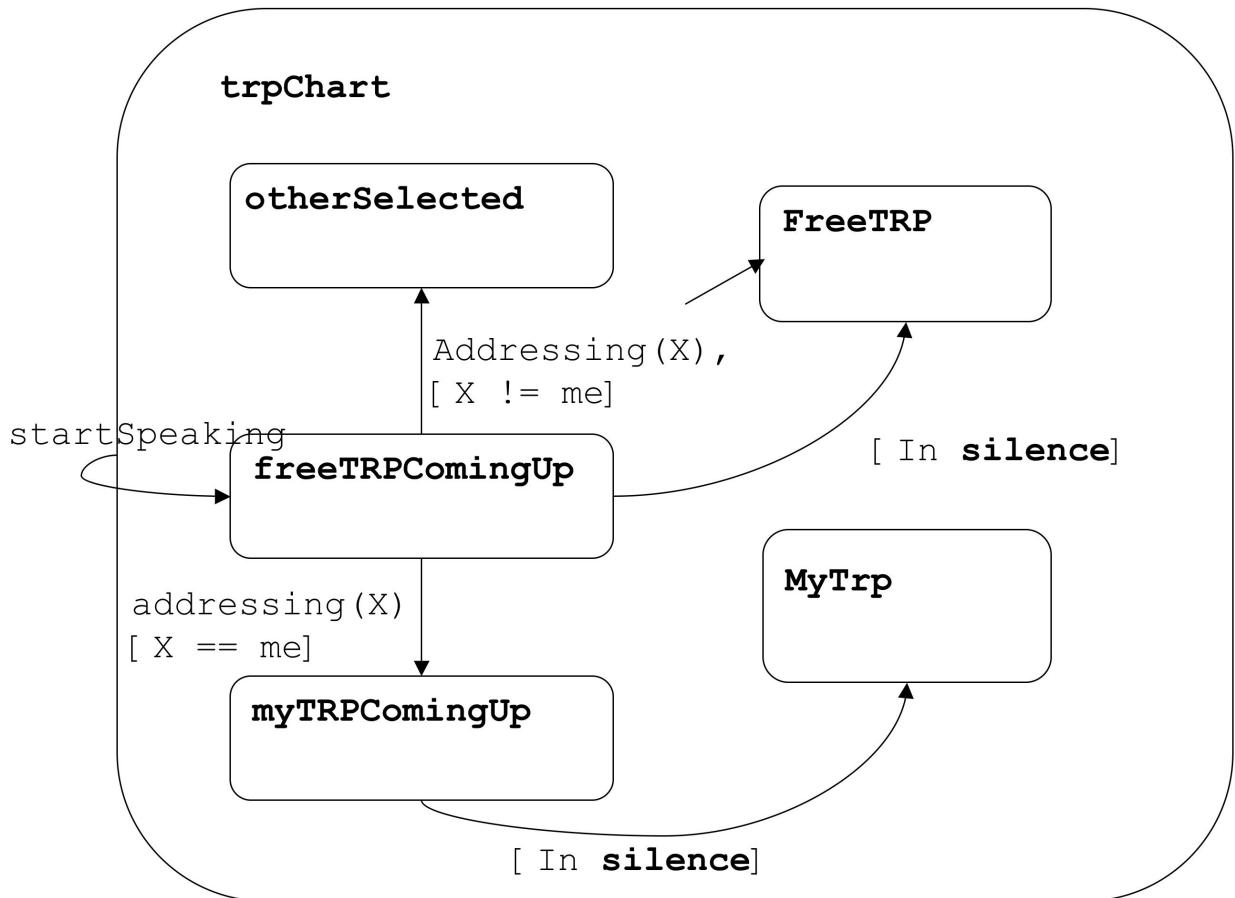


Figure 3.2: The TRP State Chart

## 3.4 A Multi-party Turn Manager

### 3.4.1 Output requirements

In Section 3.2 we specified the output we require from the turn manager. The list of output items, which is repeated below, can be further refined.

1. When this agent is the only DP (Dialogue Participant, or Dialogue Partner) with the right and obligation to speak at this TRP.
2. When an agent other than this agent is the only DP with the right and obligation to speak at this TRP.
3. When anyone (including this agent) may self-select at this TRP.
4. When this agent's current contribution is overlapping someone else's.
5. When the abovementioned overlapping has been resolved.

The items 1 and 3 concern the dialogue states when this agent has the opportunity to take the turn, either because this agent has been selected by the current speaker, or because no-one else has been selected by the current speaker. In the case of a free TRP, an agent has the opportunity to take the turn to make a contribution. In the case where this agent is selected as the next speaker, the agent has an *obligation* to take the turn. In both cases the initial action should be the same – to check whether the agent has something relevant to say. In order to simplify interaction with the dialogue manager, we wrap the two states into one superstate `openTRP`. We emit the events `openTRP.my` and `openTRP.free` when entering the substates `freeTRP` and `myTRP`, respectively.

Item 4 may be split into two different kinds of overlap. Either someone else is starting to speak when this agent is speaking, or this agent is starting to speak when someone else is trying to speak. We introduce the event `interruptionAttempt` for the case when someone else is starting to speak when this agent is already speaking and the event `missedTRP` for the other case. The states and events serving as output of the turn manager are listed in table 3.3.

### 3.4.2 Architecture

We identify three building blocks in our turn manager. One block deals with the other participants speaking states – are they speaking or are they silent? – which we call the

| Name                | Meaning   | Type  |
|---------------------|---|-------|
| openTRP.free        | Anyone may self-select at this TRP                                    | event |
| openTRP.my          | This agent is selected as the next speaker at this TRP                | event |
| othersTRP           | Someone else is selected as the next speaker                          | state |
| interruptionAttempt | Someone else is starting to speak when this agent is already speaking | event |
| missedTRP           | This agent is starting to speak when someone else is already speaking | event |
| overlap             | Speech from this agent and another DPs is overlapping                 | state |
| overlapResolved     | Speech is no more overlapping   | event |
| openTRP             | This agent may speak at this TRP                                      | state |

Table 3.3: Output expected from the turn manager

Outside chart. The second block deals with the relation between the other agents and this agent (the agent whose perspective we are taking) – is this agent talking at the same time as some other agent? We will call the chart the Inside chart. (The Outside and Inside charts correspond to the Overlap chart of section 3.3). The last chart deals with identifying TRPs and signalling them to some kind of dialogue manager/sequencer. We call this chart the TRP chart. The hierarchical nature of the formalism chosen, together with the concurrency support (orthogonality) allows us to easily model the three components inside one single statechart.

### 3.4.3 The Outside Chart

The Outside chart is supposed to keep track of “the world outside”, i.e. if the other DPs are speaking or if they are silent.

The simplest solution to the problem would be to have  $N + 1$  states ( $N = \text{number of DPs}$ ): `othersSilent`, `1Speaking`, `2Speaking`, ..., `N+1Speaking`. This would be impractical, since we then would need to grow or to shrink the state chart to keep track of the changes in the dialogue context (DPs joining or leaving the dialogue). Instead, we will use an abstraction – we will have one state `silence` and one state `speaking`, since these are the crucial states for our purposes. We will use the datamodel for storing a set which we will use to keep track of how many of the DPs are actually speaking at a given point in time. DPs are added to the set `Speakers` when they start speaking and removed from it when they stop

speaking.

The initial state is the `silence` state. If someone (except this agent) starts speaking, we will need to make a transition to the `speaking` state. We will also need to add the speaker to the set of `Speakers`. If someone starts speaking when the `speaking` state is active, it will remain active, but as we need to add the speaker to the set of `Speakers`, we will need a transition, labelled `startSpeaking(X) [X != me] / Speakers.add(X)` (the label format being `event [condition] / action`). Since the action will be the same, no matter what state is active, we add a transition from the superstate (`outsideChart`) to `speaking`, labelled as described above. This means that whatever state we are in when the event/label combination occurs, we will make the transition to the `speaking` state.

When some other agent stops speaking, we will remove that agent from the `Speakers` set, and when the set is empty, we will make the transition to the `speaking` state. Therefore we add a targetless transition with `speaking` as source state. The label will be `stopSpeaking(X) [X != me] / Speakers.remove(X)`. We also add a second transition from `speaking` to `silence`, labelled `[Speakers.isEmpty()]`.

The Outside chart is shown in Figure 3.3, and the SCXML source can be seen in Listing 3.3. The statechart is almost identical to the SilenceDetector of Chapter 1, the difference being that we use a set of agent identifiers instead of an integer to keep track of the current speakers. It may prove useful to know what agents are speaking at a given point in time.

### 3.4.4 The Inside Chart

The role of the Inside chart is to keep track of this agent's relations to the other agents present. When is this agent speaking at the same time as some other agent? Who started speaking first – is someone trying to interrupt this agent, or is it the other way around? Is the interruption legitimate?

Let us start with a very simple statechart to keep track of the simple fact whether this agent is speaking or is silent. The code for this statechart is shown in Listing 3.2. We start out with the two states `iAmSilent` and `iAmSpeaking`. We add transitions between the states labelled with the events `startSpeaking` and `stopSpeaking` respectively (and the condition that `Eventdata.speaker` evaluates to the identifier of this agent) in the obvious directions.

We are interested in the situations when this agent is speaking at the same time as some other agent (or agents). In other words, everything that we are interested in happens when this machine is in the state `iAmSpeaking`. Therefore we add the states `noOverlap` and `overlap` as substates to `iAmSpeaking`. We indicate `noOverlap` as the initial state.

```

<state id="outsideChart" target="silence">
  <datamodel>
    <data name="Speakers" expr="{New Set init}"/>
  </datamodel>
  <transition event="startSpeaking"
    cond="Eventdata.speaker != AgentName"
    target="speaking">
    <assign
      name="Speakers"
      expr="{Set.add Speakers Eventdata.speaker}"/>
  </transition>
  <state id="silence"/>
  <state id="speaking">
    <transition pcond="{Set.isEmpty Speakers}"
      target="silence"/>
    <transition
      event="stopSpeaking"
      cond="Eventdata.speaker != AgentName">
      <assign
        name="Speakers"
        expr="{Set.del Speakers Eventdata.speaker}"/>
    </transition>
  </state>
</state>

```

Listing 3.1: SCXML describing Figure 3.3

```

<state id="insideChart" target="iAmSilent">
  <state id="iAmSilent">
    <transition event="startSpeaking"
      target="iAmSpeaking"
      cond="Eventdata.speaker == AgentName"
      />
  </state> <!-- iAmSilent -->
  <state id="iAmSpeaking" target="noOverlap">
    <transition event="stopSpeaking"
      target="iAmSilent"
      cond="Eventdata.speaker == AgentName"/>
  </state> <!-- iAmSpeaking -->
</state> <!-- insideChart -->

```

Listing 3.2: The first Inside chart

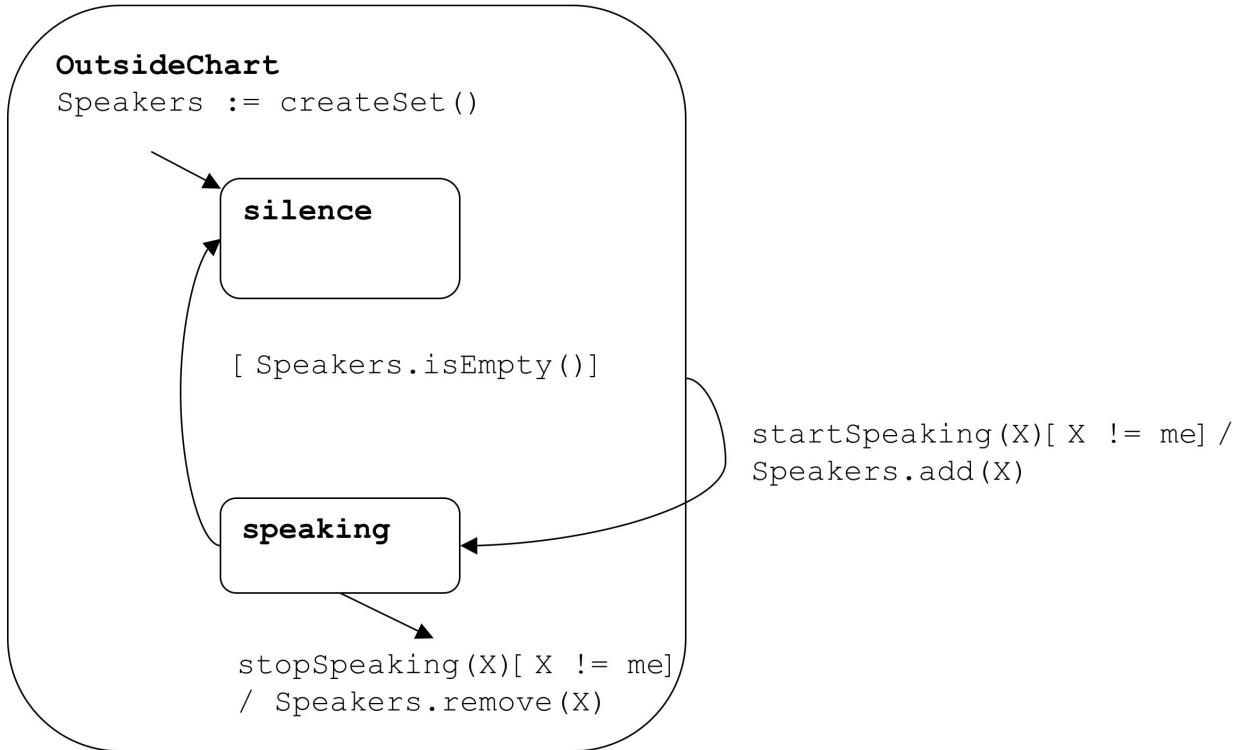


Figure 3.3: The Outside Statechart

The requirement for reaching the state `overlap` is that someone else is speaking. Hence we add a transition from `noOverlap` to `overlap` labelled with the event `startSpeaking` and the condition `[Eventdata.speaker AgentName]` meaning that the transition will be carried out if someone else starts speaking. If the state `iAmSpeaking` is active and this transition is carried out, then this agent is being interrupted. According to the refined requirements in the beginning of Section 3.4.1, we should then send the event `interruptionAttempt`. The SCXML code for the transition is shown in Listing 3.3.

```

<transition event="startSpeaking"
            cond="Eventdata.speaker \= AgentName"
            target="overlap">
    <send event="interruptionAttempt"/>
</transition>

```

Listing 3.3: The Interruption Transition

The requirements also tell us that the event `missedTRP` should be emitted when this agent has started speaking when someone else is already speaking. This can be detected by another transition between the same states, but with different transition conditions. If

we add such a transition with the transition condition [*In speaking*] after the other transition (in document order), it will only be considered when the other transition does not apply, and will therefore only apply when *someone was already speaking* when this agent started speaking. The SCXML code for the Missed TRP transition is to be found in Listing 3.4.

```
<transition cond="[{In speaking}]"
    target="overlap">
    <send event="missedTRP"/>
</transition>
```

Listing 3.4: The Missed TRP Transition

Since exiting *overlap* means that the overlap has been resolved (one way or the other), we add the sending of the *overlapResolved* as an *onExit* action in the *overlap* state. The Inside Chart can be seen in Figure 3.4

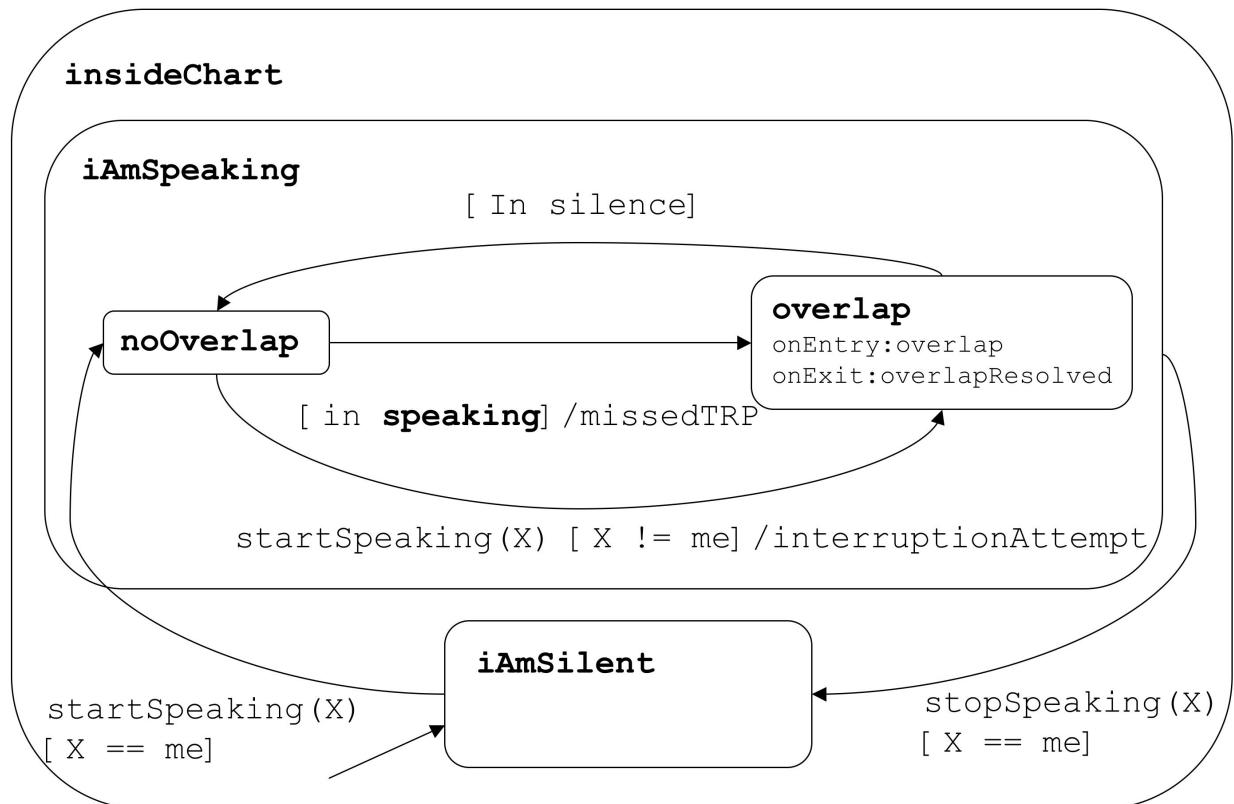


Figure 3.4: The Inside Statechart

### Overlap resolution

There are cases when the “first-starter goes” principle of conflict resolution is not applicable. Sacks et al. (1974) mentions that “problems of understanding” is prioritised over other utterances, which means that even if speaker  $S_1$  starts speaking before speaker  $S_2$ , speaker  $S_2$  will still get the turn if his or her TCU consists of negative feedback. It is not clear if this only concerns negative feedback regarding understanding, or if it is generally applicable for all kinds of negative feedback.

We can add a rule to emit an event when this happens. Assuming that the agent signals a recognised dialogue move using events on the form

```
says(move:icm#(Type#Pol) speaker:S ...)
```

and that feedback is classified as in (Larsson, 2002) we will set up an internal transition in the `overlap` state in the following way: The event condition will be `says`, and the conditions on the event data will be

```
Eventdata.move=icm#(Type#neg).
```

An event `negativeFeedbackOverlap` will be generated. The consequences of adding this transition is that if there is an overlap where this agent is involved, and if the overlapping dialogue move is negative feedback on some level, the event `negativeFeedbackOverlap` is sent. SCXML source code of the transition can be found in Listing 3.5.

```
<transition event="says"
            pcond="Eventdata.move = icm#(Type#neg)">
    <send event="negativeFeedbackOverlap"/>
</transition>
```

Listing 3.5: The negative feedback transition

### 3.4.5 The TRP Chart

We will use the TRP Chart described in Section 3.3.2 as a starting point for the TRP management in the multi-party turn manager. The basic structure will be the same, but some states and transitions will be added.

The machine in Section 3.3.2 will start out in a state called `freeTRP`, and as soon as someone starts speaking we will leave the `freeTRP` state for the state `freeTRPComingUp`.

Selection of this agent (signalled with the event `addressing(addreree:X)`), will cause the machine to move to another set of states, `myTRPComingUp` and `myTRP`. Selection of another agent causes a state `otherSelected` to be active. As soon as someone starts speaking, the state `freeTRPComingUp` becomes active.

There are two problems with this TRP chart. First, one agent can hold the turn forever as the TRP chart can get stuck in the `otherSelected` state. Second, there is a problem with the emission of events for signalling TRPs which occur after this agent's utterances. We present the problems and their respective solutions in the following paragraphs.

Say that agent *A*, equipped with our turn manager, and some other agent *B*, are engaged in dialogue. *A* asks “What time is it?” and *B* does not reply. Transcribed to a list of input events it will look like `startSpeaking(A)`, `addressing(B)`, `stopSpeaking(A)`. At this point in dialogue, *A*'s TM is in state `otherSelected`. The only way to move the machine out of the state is to read an event `startSpeaking` from the event queue.

Say that *B* did not understand the question, or did not hear it, or maybe did not perceive that somebody was trying to say something or, even, heard the question but deliberately did not answer it. In this case, we will not read a `startSpeaking(X)` event simply because no-one will start speaking.

This may be solved either internally in the TRP Chart by adding a `othersTRP` state and timers that will trigger transitions from `myTRP` to `othersTRP` respectively to the `freeTRP` state, or external to the TM, for instance elsewhere in the Dialogue Manager. We will select the first solution, and the TRP Chart with the necessary changes is shown in Figure 3.5.

The TRPs generated by this agent are generally treated in a rather sloppy way. Since the `silence` state being active is the condition triggering the TRPs, and since the speaking activity of this agent is not considered by the `outsideChart`, such TRPs are signalled too early. A simple remedy would be to add the condition that `iAmSilent` should be active for all such transitions in `trpChart`.

According to the rules of SSJ, the latest speaker should give the other DPs a chance to take the turn at a free TRP. If they decide not to, the latest speaker may go on for another turn of talk. This is not supported by this turn manager, we consider it the responsibility of the individual agent.

### 3.4.6 A Note on Projectability

The SSJ model argues that TCUs have predictable endpoints, and that human speakers can predict the endpoints (TRPs) with relatively high precision. This is supported by the

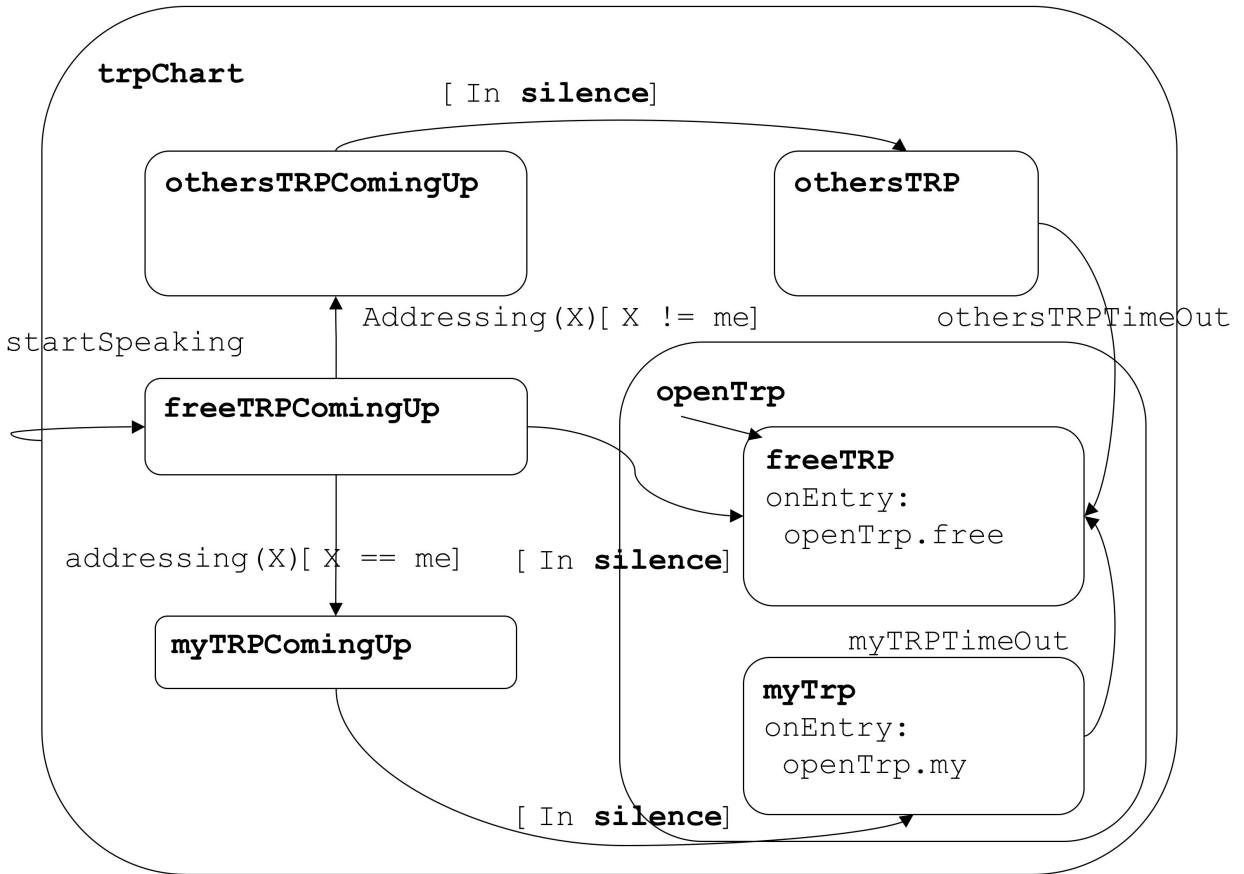


Figure 3.5: The improved TRP Chart

relative high frequency of brief overlaps at TRPs – speakers are starting to speak slightly before the previous speaker has stopped speaking in order to get the turn according to the “first-starter goes”-principle.

The turn managers presented in this chapter have no support for this kind of prediction. An attempt to include projectability into the model is presented in (Kronlid, 2006). We believe however that the best solution is to leave the TCU endpoint detection outside the turn manager, to let an external device emit `stopSpeaking` events when we are close enough to a TCU endpoint. Such a device is `/nailon/` (Edlund et al., 2005). In the SSJ model, syntactic cues are of great importance, while `/nailon/` uses prosodic cues only to detect TCU endpoints.

## 3.5 Related Work

### 3.5.1 Turn Taking Simulations

(Padilha and Carletta, 2002; Padilha, 2006) is another implementation of the SSJ turn management model. However, the aim for (Padilha and Carletta, 2002; Padilha, 2006) is not to build a component to be used in a dialogue system, but to simulate a conversation with adequate turn distribution among the interlocutors, turn length etc. The model consists of number of agents, taking turns talking (simulated talk). They communicate via a blackboard, sending signals for TRPs, pre-TRPs, backchannel communication etc., and the behaviour of the agents are based on probabilistic decisions. The focus in (Padilha, 2006) is to create a turn management model that is as close to human turn-taking as possible, while our focus is to build a robust, flexible turn manager which is compatible with human turn taking.

The aim of (Hulstijn and Vreeswijk, 2003) is to evaluate agent programming languages by using turn-taking as a case. The participants are modelled as agents, which use probabilistic decisions for generating behaviours such as keeping the turn, starting to speak etc. The model includes projectability of TRPs, using an explicit language model, where all the valid utterances in the agents' language are stored. The utterance that has been perceived so far is then compared to the strings in the language model to estimate how far away the next TRP is.

### 3.5.2 Thorissón

Kristinn Thorissón presents a two-party turn manager in (Thórisson, 2002). Among its merits is the joining of the two major approaches to turn management, the signal-based approach (Duncan and followers (Duncan, 1972)) and the opportunities-based approach (Sacks, Schegloff and Jefferson and followers). He sees the SSJ model as the basic rule system governing turn management, but criticises it for ignoring anything but the audio signal (e.g. gestures, gazes, posture etc.) He then suggests that the work of Duncan and followers fill these empty spaces in the SSJ model, and presents a state based turn manager where the two models are merged.

A number of behaviours associated with turn taking is described in terms of “trigger conditions” and “restore conditions”. If all the trigger conditions are fulfilled, the behaviours are executed. They cannot trigger again until the restore conditions have been fulfilled. Some of these descriptions are as follows:

Name: Hesitate-during-delay-in-formulation

Behaviour: Show-hesitation

Trigger conditions:

```
Dialog-on
AND I-have-turn
AND Speech-data-available-from-other
AND Time-since(Other-is-speaking) > 70 ms
AND NOT I-have-reply-ready
AND NOT Other-is-speaking
```

Restore conditions: I-give-turn

Name: Pay-attention-to-my-own-action

Behaviour: Turn-head-to(workspace)

Trigger conditions: Im-executing-topic-realworld-task

Restore conditions: NOT Im-executing-topic-realworld-task

Name: Acknowledge-other-is-addressing-me

Behaviour: Turn-to(other)

Trigger conditions:

```
Im-executing-topic-realworld-task
AND Other-is-looking-at-me
AND other-is-facing-me
AND other-is-speaking &
```

Restore conditions: NOT Im-executing-topic-realworld-task

Name: Show-Im-done-with-task-by-looking at other

Behaviour:

```
Turn-head-to(other)
AND Gaze-at(other)
```

Trigger conditions: NOT Im-executing-topic-realworld-task

Restore conditions: Im-executing-topic-realworld-task

Looking at the formalisations and at the flexible and transparent dialogue manager that has been presented in this thesis, it seems like most of the behaviours should be possible to integrate in the the dialogue manager, given that one can obtain the fine grained information about other agents' posture, gaze, etc. that is required to trigger some of the behaviours. This may prove to be expensive to do in the real world, but in a virtual world, it could be affordable.

## **3.6 Summary**

In this chapter, we have described the motivations for, and the construction of, a turn manager for conversational agents operating in a multi-party environment. We have shown that a specification using Harel statecharts is possible, and that a transition from the theory described by Sacks et al. (1974) to a state-based implementation is straightforward. The result is a highly modular statechart with three simple substates or modules, each of them dealing with one well-defined part of the turn taking problem.

The turn manager will be used in Chapter 4, where the aim is to build a dialogue manager for conversational agents capable of engaging in multi-party dialogue.

# Chapter 4

## Multi-Party Dialogue Management

### 4.1 Introduction

The aim of this thesis is to build conversational agents, capable of multi-party dialogue, for use in games, information services etc. In this chapter, we will create the theoretical background for an implementation of a dialogue manager working in such a context. We will assume that turns are managed in the SSJ manner.

The resulting dialogue manager will be able to handle what we will call *distributive* and *collective* questions. It will be able to generate and interpret agreement and disagreement, and also acceptance and rejection of assertions and questions. It will also identify and handle TRPs where a set of speakers, of which the next speaker should be a member, is selected. The dialogue manager will be named *Multi-IBiS*.

The dialogue manager described in Chapter 2 was an SCXML implementation of the dialogue manager IBiS1 (Larsson, 2002), based on the concept of issue-based dialogue management (IBDM), a development and implementation of Ginzburg's KOS (Ginzburg, 1996). Ginzburg and Fernández (2005a,b) present an extension of the “standard” QUD protocols, to cover also multi-party dialogue. Ginzburg and Fernández use the conversation roles defined by Erving Goffman (1981) as a starting point for the new protocols. We will take a closer look at these roles, and also at the similar, but richer, description of such roles by Clark and Carlson (1982). This investigation leads us to believe that the protocols of Ginzburg and Fernández need some modifications in order to handle multi-party dialogue.

Traum and Rickel (2002) describes a dialogue system for a multi-party environment, fea-

turing both QUD/IBDM components and an obligations layer, based on (Matheson et al., 2000) and (Allwood, 1994). More details about the system and the thoughts behind it is given in (Traum, 2003b). We argue that the obligations as such are necessary, but that the obligations layer is unnecessary in the simple systems explored in this thesis, and that the obligations can be encoded in the information state update rules and in the QUD structure itself.

## 4.2 Background

In the background we will review some literature in order to establish some important concepts which will be used in this chapter. We will start by introducing the conversation roles from Goffman and from Clark & Carlson, respectively. From the latter, we will also borrow the “principle of responsibility”. We will continue with Rickel & Traum, and the concept of obligations in connection to questions and dialogue. Finally, we will examine the “multilogue” utterance processing protocols suggested by Fernández and Ginzburg, which we will use as a blueprint for our implementation (or specification) of a multi-party dialogue manager.

### 4.2.1 Goffman

Erving Goffman discusses the different roles taken and assigned in a conversation in (Goffman, 1981), and on his analysis there are (broadly speaking) three kinds of listeners: *overhearers*, *addressees* and other *ratified participants*.

Overhearers are participants, whose unratified participation may or may not be known and encouraged. Overhearers are not taking part in the interaction, just observing it. The ratified participants are divided into two groups: those who are addressed and those who are not. The addressees are “oriented to by the speaker in a manner to suggest that his words are particularly for them, and that some answer is therefore anticipated from them”.

### 4.2.2 Clark & Carlson

The purpose of (Clark and Carlson, 1982) is to extend the standard speech act theory to handle multi-party interaction. The approach involves introducing the *informative* speech act, used as a means to describe how information flows from the speaker, not only to the direct addressee, but to all the DPs present in the interaction. The basic idea is that the

informative speech act conveys information to all participants in a conversation about the illocutionary speech act that the speaker is carrying out towards the addressee(s).

(Clark and Carlson, 1982) also introduce the *principle of responsibility* – meaning that, generally, each participant (in the hearer role) of a conversation is responsible for keeping track of what has been said, and (in the speaker role) for enabling everyone else to keep track of what is being said.

Clark and Carlson list a number of roles in a conversation. There is a *speaker* and a number of *hearers*. The hearers of an utterance are divided into *participants* and *overhearers*. The group of participants are subdivided into *addressees* and *side participants*. The group of overhearers are subdivided into *known* and *unknown* overhearers. Table 4.1 illustrates this categorisation of roles.

| Speaker | Hearers     |         |                   |            |
|---------|-------------|---------|-------------------|------------|
|         | Overhearers |         | Participants      |            |
|         | Known       | Unknown | Side participants | Addressees |

Table 4.1: The roles in spoken interaction according to (Clark and Carlson, 1982).

Another important concept in Clark and Carlson's account is *audience design* – that speakers always design their utterance with overhearers, side participants and addressees in mind. Or rather, the speaker designs his utterance in such a way, that the hearers of an utterance are assigned the correct roles. The set of roles is (as the authors point out) very similar to Goffman's.

The principle of responsibility governs the basic utterance design, in that the utterance must be designed in such a way that all the *participants* understand what the speaker is trying to convey. The speaker assigns the role of participant to all parties in the conversation – and the role of overhearer to all other hearers – by the design of his utterance.

The addressees are the targets of the speaker's utterance, and Clark and Carlson (1982) note that “most of Searle's ‘felicity conditions’ for speech acts [preconditions for the validity of a speech act] are satisfied as part of addressee design”. In a response (Clark, 1986) to criticism of (Clark and Carlson, 1982) from Allan (1986), Clark stresses that addressing is signalled by vocatives and by the illocutionary acts performed. “The vocative that is actually used or could have been used in an utterance specifies the addressees of at least one illocutionary act performed with that utterance” (Clark, 1986, p. 521). He adds that the *vocative criterion* is not sufficient, that other illocutionary acts performed in the utterance may have other addressees. Besides the vocative criterion there is the *respondent criterion*. The respondents are, for an illocutionary act the “set of people who are intended to effect certain perlocutions”. For instance if A invites B and B's family for dinner, B's family members (including B) are the respondents. “The addressees of an illocutionary

act are those respondents who are currently participating in the conversation.” (Clark, 1986, p.521). So if A’s invitation to B and his family is made when only A and B are participating in a conversation, only B is an addressee of the act.

Speakers can also design utterances with overhears in mind – in order to make them form correct, incorrect or no hypotheses of the illocutionary acts at all based on what they overhear.

By designing an utterance in a certain way, a speaker can address a hearer without knowing what particular hearer he addresses. As an example, (Clark and Carlson, 1982) uses a setting where George is on a roof, out of sight of Alistair and Fergus. He is then saying “Hand me the saw”, without knowing which of them has the saw. By his utterance, Clark and Carlson (1982) claims that George is addressing the person, Alistair or Fergus, who has the saw in his hand. The one with the saw is the only participant who is a respondent to the utterance. The process is named *addressing-by-attribution*.

We will not keep the very detailed and clear-cut notion of addresseehood suggested by (Clark and Carlson, 1982). Instead we will use a more pragmatic notion, where the addressee is the DP who is selected as the next speaker by the current speaker. We will however use the term “addressing-by-attribution” to describe what happens when a group of DPs is addressed with a question that only some of the addressees can answer.

#### 4.2.3 Traum & Rickel

In (Traum and Rickel, 2002; Traum, 2003b), David Traum and Jeff Rickel describe a dialogue system featuring multi-party dialogue management based on QUD and an obligations layer based on (Matheson et al., 2000).

In this implementation, QUD is used for question management – to keep track of which questions are currently under discussion in the conversation and to resolve elliptical answers to questions. The obligations layer is used to manage the obligations which arise in dialogue e.g. the obligation for a DP to answer a question directed to it, the obligation to ground an utterance. According to (Traum, 2003b) neither QUD nor obligations alone can manage this – a combination is needed.

In the view of (Traum, 2003b), QUD is a part of the structure of conversations rather than an independent relation between DPs. Obligations, on the other hand, are “aspects of the social fabric between agents”, and it is stated that obligations persist over conversation boundaries.

In (Traum, 2003b), it is pointed out that the suggested model leaves a number of unresolved

“complexities”, such as:

- when to see a question as resolved (and removed from QUD)
- when to answer a question on QUD that someone else (or no-one) is obliged to answer
- whether an answer by one agent resolves the obligation for another agent to answer.

Ginzburg and Fernández (2005b) criticises the multi-party dialogue manager of (Traum and Rickel, 2002) by saying that its grounding model only allows parallel dialogues, not true multi-party dialogues. The reason is, according to (Ginzburg and Fernández, 2005b), that the grounding model only works in cases where there is one single initiator and one single responder. However, there is a mention of “one or more responders” in the description of the “common ground unit” (Traum, 2003b), but there is no answer to the question, posed in (Ginzburg and Fernández, 2005b), what the grounding model should look like when there are multiple responders: “should the contents be considered grounded when any of the addressees has acknowledged them? Should evidence of understanding be required from every addressee”?

#### 4.2.4 Multilogue QUD

Previous work on QUD by Jonathan Ginzburg has been focused on two-party dialogue (see Chapter 2), but Ginzburg and Fernández (2005a,b) presents an attempt to scale up the dialogue protocols to cover also multi-party dialogue. They introduce the hearer roles Overhearers, Side participants<sup>1</sup> and Addressees (described in Section 4.2.1) into the model.

##### Overhearers

The overhearers observe the interaction, but “their context is not facilitated for them to participate” (Ginzburg and Fernández, 2005b), which means that they may or may not understand the interaction. The protocol for the overhearers is the simplest one:

*Given a dialogue protocol  $\pi$ , add roles  $C_1, \dots, C_n$  where each  $C_i$  is a silent participant: given an utterance  $u_0$  classified as being of type  $T_0$ ,  $C_i$  updates  $C_i.DGB.FACTS$  with the proposition  $u_0 : T_0$ .*

---

<sup>1</sup>Side participant does not seem to be a term used by Goffman, but borrowed from (Clark and Carlson, 1982). However, the term fits into Goffman’s taxonomy.

What this means is that whenever one of the overhearers hears an utterance (or observe a contribution)  $u_0$ , he or she (or it) registers the occurrence of the contribution/utterance and its type<sup>2</sup>, adds it to the *FACTS* part of the agent’s dialogue game board. This is done without any processing of the content of the utterance (although (Ginzburg, 2007) indicates that the principle was designed to allow for interpretation). Stated less formally, an agent only records that some particular utterance with some particular features has been uttered.

## Side Participants

The side participants are taking part in the dialogue, but they are not currently speaking or being addressed. A side participant is a listener, not obliged to take any verbal action in response to a contribution. The modifications to the standard utterance processing protocols (see Section 2.3.1) are the following.

*Given a dialogue protocol  $\pi$ , add roles  $C_1, \dots, C_n$ , which effect the same contextual update as the interaction initiator.*

We believe that the intuition behind the principle is that the side participants are silent observers, and after uttering the move, the initiator is a silent participant. Note that the wording is “contextual updates” – which excludes the part about releasing the turn.

Here is the *Assertion* protocol for a conversation involving the set of DPs  $\{A, B, C_1, \dots, C_n\}$ , where  $A$  is the latest speaker,  $B$  is the addressee of the utterance and  $C_i$  are side participants:

---

<sup>2</sup>The type is here a characterisation of the utterance in terms of syntax, semantics and phonology (Ginzburg, 2007)

```

LatestMove = Assert(A,p)
A:   push p? onto QUD;
      release turn
C_i: push p? onto QUD;
B:   push p? onto QUD;
      take turn;
      (Option 1: Accept p,
       Option 2: Discuss p?)

```

```

LatestMove = Accept(B,p)
B:   increment FACTS with p;
      pop p? from QUD;
C_i: increment FACTS with p;
      pop p? from QUD;
A:   increment FACTS with p;
      pop p? from QUD;

```

The protocol allows, according to (Ginzburg and Fernández, 2005b), for “communal acceptance” – one person accepting a proposition on the behalf of a larger group. The idea is to let one DP,  $B$  take the role of “spokesperson” for the group, consisting of the side participants  $C_1, C_2, \dots, C_n$ .

The *Querying* protocol for a conversation involving the same set of DPs  $\{A, B, C_1, \dots, C_n\}$  is shown here:

```

LatestMove = Ask(A,q)
A:   push q onto QUD;
      release turn
C_i: push q onto QUD;
B:   push q onto QUD;
      take turn;
      make max-qud-specific utterance.

```

As we can see, the side participants perform the same updates as the speaker, pushing the question  $q$  on QUD. They should not take the turn for making a max-qud-specific utterance.

## Duplicate Responders

In Goffman's description, answers are expected from addressees. However, the wording "more so than from other participants" suggests a possibility for side participants to answer questions.

To handle cases where more than one DP answers a question, Ginzburg & Fernández introduce the ADR principle – *Add Duplicate Responders*<sup>3</sup> – to cover such cases. Following Goffman's and Clark and Carlson's taxonomy of conversational roles, we would have expected the principle to be named "duplicate addressees" instead of "duplicate responders". We will return to this issue in Section 4.4.1.

*Given a dialogue protocol  $\pi$ , add roles  $C_1, \dots, C_n$  which duplicate the responder role.*

The protocol resulting from applying the ADR principle to the standard querying protocol looks like this – Querying with multiple responders:

```

LatestMove = Ask(A,q)
A: push q onto QUD; release turn
B: push q onto QUD;
    take turn;
    make max-qud-specific utterance;
    release turn
C_1: push q onto QUD;
    take turn;
    make max-qud-specific utterance;
    release turn
C_2: push q onto QUD;
    take turn;
    make max-qud-specific utterance;
    release turn
...
C_n: push q onto QUD;
    take turn;
    make max-qud-specific utterance;
    release turn

```

---

<sup>3</sup>The use of the term 'responder' in (Ginzburg and Fernández, 2005a,b) seems to differ from the use of the same term in (Clark and Carlson, 1982). In the former case, a 'responder' seems to be a DP who is currently carrying out the act of responding. In the latter case the 'responder' is the target for a perlocutionary act.

Informally stated, the cooperative querying protocol for duplicate responders works like the following: After an *Ask* move with content  $q$  has been made by  $A$ ,  $A$  pushes  $q$  on QUD and releases the turn. At the same time (probably) as  $A$  pushes  $q$  on QUD,  $C_1, C_2, \dots, C_n$  and  $B$  also push  $q$  on QUD. Then, in an order that is not determined by the protocol, each one of  $C_1, C_2, \dots, C_n$  and  $B$  takes the turn, makes a max-qud-specific utterance (see Section 2.3.1) and releases the turn.

The corresponding assertion protocol – Assertion with multiple responders – looks like this:

```

LatestMove = Assert(A,p)
A:      push p? onto QUD;
        release turn
B: push p? onto QUD;
    take turn;
    (Option 1: Discuss p?,
     Option 2: Accept p)
C_1: push p? onto QUD;
    take turn;
    (Option 1: Discuss p?,
     Option 2: Accept p)
C_2: push p? onto QUD;
    take turn;
    (Option 1: Discuss p?,
     Option 2: Accept p )
...
C_n: push p? onto QUD;
    take turn;
    (Option 1: Discuss p?,
     Option 2: Accept p)

```

Each of the addressees need to push the question  $p?$  on QUD and to take the turn to either accept  $p$  or discuss  $p?$ .

The result of applying ADR is a multi-party dialogue where every addressee needs to respond to a question (or to comment on the other parties responses), where every addressee needs to show evidence of understanding for an utterance to be considered grounded, and where every DP needs to address or explicitly accept an assertion.

## 4.3 Preliminary Discussion

### 4.3.1 Some words on Utterance Processing Protocols

There are some issues with the utterance processing protocols. Let us take a look at Ginzburg’s “cooperative querying protocol” (which was discussed in Section 2.3.1):

1. LatestMove.Cont = Ask(A,q): IllocProp
2. A: push q onto QUD; release turn
3. B: push q onto QUD; take turn; make q-specific utterance

Despite the formal nature of the protocol, it is unclear *when* things happen. Is the description sequential or parallel? The numbering suggests sequence, and so does the division of the lines in “statements”. But it cannot be that A first pushes q on QUD and then releases the turn – before B pushes q on QUD. The protocols are blending sequence and parallelism, without clearly stating which is which.

### 4.3.2 Individual, Distributive and Collective Questions

Looking at questions in a multi-party setting, and the ways that they are answered, there seems to be a number of different answering behaviours. One of the behaviours is the simple answering of a question by one DP. A question is asked by one DP, and answered by another one, who is the only addressee of the question. We will call such a question a *individual* question. See Example 4.1.

A: Do you want some coffee?

B: No thanks.

Example 4.1: Individual question on the coffee issue

Another kind of behaviour is when a question is addressed to more than one DP, and it is answered by one of the addressees, and the other addressed DPs either agree or disagree with the answer. We call this a *collective* question. See Example 4.2. Other addressees may also give complementary information, in such a way that an answer is collectively formed. Example 4.3 shows such an interaction.

A: What do you think -- coffee now or later?

B: Now

C: Yeah

Example 4.2: Collective question on the coffee issue

A: What did he look like?

B: Tall with long, brown hair.

C: He had black denim jeans and some kind of strange hat.

D: Yeah, and a greyish army jacket.

Example 4.3: Collective question on the issue of how to identify the suspect.

A third kind of behaviour is when a question is directed to more than one DP, and it is answered “individually” by all of the addressees. We call this kind of questions *distributive*<sup>4</sup>. An example is given in Example 4.4.

A: Would you like some coffee?

B: Yes, please!

C: No, thanks.

Example 4.4: Distributive question on the coffee issue

It seems like the question in itself plays a role in deciding if it should get the distributive or collective reading, but contextual factors (activity, number of addressee's, social status etc.) are also important. It seems, for instance, like the question in 4.1 cannot get a collective reading, when directed to B and (another DP) C, without changing the wording to something like “Does B want some coffee”.

Our hypothesis is that the choice between a collective and a distributive reading is determined by whether the question is about a relation between each individual in the addressee set and something in which case the reading will be distributive, or if the question is about a relation between a collective (including some or all of the individuals in the addressee set) and something in which case the reading becomes collective.

For instance, a question regarding travel destination can be collective or distributive depending on whether the respondents are travelling as a group or not. If the question is

---

<sup>4</sup>The notion of distributive and collective questions is similar to the distributive and collective *requests* of Clark and Carlson (1982). When a collective request is made (to more than one addressee) the addressees are expected to fulfill the request in a coordinated manner. Clark and Carlson (1982) use an example of Noah requesting a certain coordinated knife-juggling trick from Shem and Ham: “Begin the trick now”. An example of a distributive request is “close your eyes”, which requires the addressees to close their eyes individually. The distinction between distributive and collective is also done in the literature of plural noun phrases (e.g. (Gillon, 1996)).

asked by someone working at a travel agency, the *intended* reading can be either collective or distributive, or the question can be used to find out if the respondents are travelling as a group or not, depending on what kind of answers they are giving.

Traum (2003b) raises the issue of whether an answer by one agent to a question resolves the obligation for another agent to answer it. Our account of questions opens for a solution where the dialogue manager have two distinct cases to handle: individual and collective questions, where an answer by another agent does not resolve the obligation to answer them, and collective questions, where the obligation is resolved by an answer from another agent). The issue has been transformed into the problem of correctly identifying the question types.

### 4.3.3 Challenging the Multilogue Protocols

We will now return to the multilogue principles and protocols of Section 4.2.4. All three of the principles above needs to be applied to the utterance processing protocols to achieve the possibility of full multi-party dialogue. However, there are some issues regarding responsibility in connection to the protocols.

#### ADR and collective questions

Is it really true that a speaker addressing a group of people expects an answer (or confirmation) from all the addressees? That is the result of applying ADR to the querying protocol. Consider the dialogue in Example 4.5, where a number of DPs are asked a collective question on the time issue. To us, it looks a bit odd. Even if all the answers of C, D and E were replaced with non-verbal acceptances of B's answer, it would probably be too much under most circumstances (although one can think of situations where it would be appropriate, such as when the issue of what time it is is of great importance). Ginzburg and Fernández (2005b) mentions the lack of capacity of the speaker to monitor all the listeners for feedback as a possible reason for non-existing feedback in these cases.

One solution would be to use the ASP version of the querying protocol instead, which would license a kind of “communal responding”, where one of the addressees behaves as a spokesperson for the group of addressees. Such a dialogue is shown in Example 4.6. The dialogue seems plausible, but the fact that *A* addresses the group with the vocative “Hey Guys”, in our opinion, rules the option to use the ASP protocol out, according to the *vocative criterion* of (Clark and Carlson, 1982) – which states that a vocative indicates the addressees for at least one illocutionary act in the utterance. In our opinion C, D and E cannot be considered Side Participants – they are addressees. One could argue that this is

DPs: {A, B, C, D, E}

A: Hey guys, do you have the time?  
 B: Yes, half past two.  
 C: Yeah  
 D: [Nods]  
 E: Mmm.  
 A: OK, thanks

Example 4.5: Asking a group of people for the time, ADR protocol

a case of “addressing by attribution”, that *A*’s first utterance should be interpreted as “Hey guys, can the one of you who know the time please tell me what time it is?” In response to this argument, we can use a dialogue where the first utterance is “Hey guys, should we take the coffee now or later”, which should not be possible to interpret as “addressing by attribution”.

DPS: {A, B, C, D, E}

A: Hey guys, do you have the time?  
 B: Yes, half past two.  
 A: OK, thanks

Example 4.6: Asking a group of people for the time, ASP protocol

However, in cases where there is no consensus on the time issue – where the question has a distributive reading – it seems like the ADR protocol does its job. Consider, for instance, the Example 4.7, where the utterance of C is an answer which offers the querier additional information, without necessarily being a correction of B’s utterance. The protocol is also applicable in the tutor/student situation, where a number of responders (students) each are expected to give one individual answer to some question from the tutor (the prototypic example of a distributive question).

DPS: {A, B, C}

A: Hey guys, do you have the time?  
 B: Yes, half past two.  
 C: My watch says twenty past.  
 A: OK, thanks

Example 4.7: Asking a group of disagreeing people for the time, ADR protocol

Ginzburg and Fernández (2005b) argue that the side participants’ assertion protocol can

be used to model “communal acceptance” – that one DP accepts an assertion on behalf of a group of DPs. The description of the side participant role in (Clark and Carlson, 1982) and the description of the corresponding role in (Goffman, 1981), suggest that there is no need for side participants to accept or reject anything. In consequence, we believe that the side participant protocol should not be used for describing communal acceptance. Communal acceptance should, in our view, be carried out by a group of addressees, not a group of side participants<sup>5</sup>.

At this stage it should be clear that there is a need for another protocol which handles collective questions. We believe that the key issue is that being an addressee (as in (Goffman, 1981)) does not necessarily mean that you have to respond, at least not in the case where you are asked a collective question. The protocol needed takes care of multiple *addressees*, as opposed to multiple *responders*<sup>6</sup>. We will present the protocol in Section 4.4.1. But first we will turn to the issue of addressing, turn management and obligations.

## Overhearers and Side Participants

In the original formulation of the Add Overhearers Principle, an overhearer is described as a completely passive hearer, only registering what is said, without drawing any conclusions from what he is hearing. In contrast, Clark and Carlson (1982) argues that a speaker can make known overhearers form certain hypotheses about the meaning of what is said. Ginzburg (2007) argues that overhearers indeed need access to “QUD info” in order to interpret the dialogue, and thus that the principle is not valid.

Clark and Carlson (1982) state that the roles are assigned by means of utterance design and that the speaker designs his/her utterance in such a way that *participants* can understand it. If this is true – what is then the difference between an overhearer and a side participant? According to our interpretation of Clark and Carlson (1982), the difference is one of responsibility. According to “the principle of responsibility”, the speaker has a responsibility towards the participants to make sure that he or she constructs the utterance in such a way that they can understand it. The side participants have, according to the same principle, the responsibility to try to follow the conversation (Clark and Carlson, 1982). Neither the overhearers nor the speaker have such responsibilities towards each other.

It seems reasonable that the responsibilities concerning understanding, shared by the speaker and the side participants, are accompanied by access to the tools needed to make sure whether such an understanding has been established or not. The coordination of

---

<sup>5</sup>Perhaps a communal acceptance (in some sense) could be carried out by side participants in a situation where someone explicitly is elected as a spokesperson for a group.

<sup>6</sup>in the sense of “DP carrying out the act of responding”

information in dialogue is commonly referred to as “grounding”. Larsson (2002) gives an issue-based account of grounding, where grounding is carried out on four different levels – contact, perception, understanding and acceptance, where each level is presupposed by the next one (acceptance presupposes understanding presupposes perception presupposes contact).

We suggest that the difference between an Overhearer and a Side Participant is that the former has no access to the grounding toolset (feedback), while the latter has access to grounding on all levels but acceptance. An addressee has access to all levels. Some of the data presented in (Eshghi and Healey, 2007) supports this view.

#### 4.3.4 Two Systems of Obligations

The example in Example 4.8 (from (Ginzburg and Fernández, 2005b)) shows a dialogue where three DPs consider themselves as addressees in the sense of (Clark and Carlson, 1982). There is one question, and all the three addressees collectively construct the answer. We cannot tell, without asking “Anon 1”, if the question is intended to have a collective or a distributive reading – if the intension of “Anon 1” is that each of the addressees should respond with their respective opinion on “finance”, or if they are supposed to come up with an “objective” answer.

```
Anon 1: How about finance then? <pause>
Unknown 1: Corruption
Unknown 2: Risk <pause dur=30>
Unknown 3: Wage claims <pause dur=18>
```

Example 4.8: Multi-party dialogue from BNC (Ginzburg and Fernández, 2005b)

According to (Sacks et al., 1974), they cannot be selected as  $speaker_{n+1}$ ,  $speaker_{n+2}$  and  $speaker_{n+1}$  by  $speaker_n$ . It *could* be that Anon 1 is selecting Unknown 1 as the next speaker, and when he is done speaking Unknown 2 self-selects etc. However it seems plausible that Anon 1 can indeed select Unknown 1–3 to respond the question. The problem is then that the SSJ model cannot account for the selection process, at least not in the way that one perhaps would like.

It is however mentioned in (Sacks et al., 1974, p.718), that “current speaker selects next techniques” can utilise “social identities” to select the next speaker from a set of DPs<sup>7</sup>. What if the techniques to select speakers from sets of DPs are not restricted to social identities?

---

<sup>7</sup>Note that it still holds that the current speaker can select only the next speaker – in this case the current speaker selects any one participant who has a certain “social identity”.

Our hypothesis is that any speaker, who addresses a number of DPs with a question, expects the DPs to, individually or collectively, come up with a satisfactory answer to the question and to “select<sup>8</sup>” a spokesperson for the DPs (the DP who will make the first utterance after the question). The addressed DPs use self-selection (or selection using gaze or gesture) within the group to select the DP who will make that utterance. After that first contribution, the whole turn-management rule-set apply. This hypothesis covers both individual and collective questions.

If we take another look at Example 4.8, this means that after the first answer by **Unknown 1** *any* DP can self select, assuming that **Anon 1** is not selecting **Unknown 2** at that point (by gaze etc.). However, it is not necessarily so that the question that **Anon 1** asked is open to answer by any participant. Compare, for instance, (the constructed) Examples 4.9, 4.10 and 4.11. We have added information about addressing to all the examples. We have selected the symbols  $\rightarrow \{\}$  to indicate the addressing information ( $A \rightarrow \{B\}$  means that  $A$  is addressing  $B$ ,  $B \rightarrow \{A, C\}$  means that  $A$  and  $C$  are addressed by  $B$ .  $A \rightarrow \{\}$  means that  $A$  addresses no-one in particular). In Example 4.9, **A 1** is addressing the same set of DPs

|  |                                 |
|--|---------------------------------|
| <b>A 1</b> $\rightarrow \{U 1, U 2, U 3\}$ : | How about finance then? <pause> |
| <b>U 1</b> $\rightarrow \{\}$ :              | Corruption                      |
| <b>U 4</b> $\rightarrow \{\}$ :              | Risk                            |
| <b>A 1</b> $\rightarrow \{U 4\}$ :           | I wasn't asking you             |

Example 4.9: Side participant answering question and being corrected

as in the original example. **U 1** offers his response as before, but then **U 4**, who was not a member of the addressed set, answers the question. In this particular case, the questioner meant that only the intended addressees should answer the question, and the “offender” is silenced by **A 1**'s second utterance. This is to be contrasted with Example 4.10, where the utterance by **U 4** was accepted.

It seems that in most cases, making the move from the group of side participants to the group of addressees is rather uncontroversial, and (Goffman, 1981) notes that “some answer is [...] anticipated from [the addressees], more so than from the other ratified participants”.

|  |                                 |
|--|---------------------------------|
| <b>A 1</b> $\rightarrow \{U 1, U 2, U 3\}$ : | How about finance then? <pause> |
| <b>U 1</b> $\rightarrow \{\}$ :              | Corruption                      |
| <b>U 4</b> $\rightarrow \{\}$ :              | Risk                            |
| <b>U 3</b> $\rightarrow \{\}$ :              | Wage claims                     |

Example 4.10: Side participant answering question.

---

<sup>8</sup>The word ‘select’ is used here in a very technical sense. No voting or other intellectual process needs to be involved.

```

A 1 -> {U 1, U 2, U 3}: How about finance then? <pause>
U 1 -> {}: Corruption
A 2 -> {U 1, U 2, U 3}: And how is sales and marketing?
U 1 -> {}: Fine
U 2 -> {}: Regarding finance risk must also
             be considered <pause dur=30>
U 3 -> {}: and wage claims <pause dur=18>

```

Example 4.11: Side participant raising new issue and addressees are returning to old issue

It seems like there are two parallel systems of addressing. We have one addressing system which governs turn management, and another one governing which DP can answer which question. We will refer to the selection of next speaker by the term *Turn Assignment*, abbreviated *TA*. We will refer to the selection of a respondent (a DP with the *right* (and possibly an obligation) to address a question) for an question by the term *Question Assignment*, abbreviated *QA*. Potentially, one could add *Action Assignment* to the list as well, for handling requests. Action Oriented Dialogue is however outside the scope of this thesis.

One can break the rules of the turn system by speaking when another person is selected as the next speaker – or by not speaking when one is selected as the next speaker. One can break the rules of addressing and the right to address questions (or question ownership) by answering a question directed to someone else. Answering someone else's question may provoke the utterance “I wasn't asking you” from the person who asked the question, and we will use the question as a test phrase for breaking question assignment.

We will now present some examples of QA and TA in order to show more in detail how the two systems of assignment works. Some examples show interactions where the assignment, others show the breaking of both, or one of the assignments. We expect the breaking of both principles to be more awkward (or impolite) than breaking a rule from one system alone. The Examples 4.12 to 4.15 show interactions where only one DP is addressed; Examples 4.16 to 4.21 show interactions with more than one addressee.

In Example 4.12, the utterance of A is directed to B only. B answers the question, and then C asks a question. Both TA and QA are respected.

In Example 4.13 C ignores the TA (the turn is assigned to B), but QA is respected, since C doesn't answer the question directed to B. It seems like the behaviour in this example is slightly less polite than the behaviour from the previous example.

Example 4.14 shows a dialogue in which TA is respected but QA is violated. A directs

A -> {B}: What do you want for your birthday?  
 B -> {}: a vase  
 C -> {B, A}: Should we go for a walk?

Example 4.12: Respects TA (Turn Assignment) and QA (Question Assignment), one addressee

A -> {B}: What do you want for your birthday?  
 C -> {B, A}: Should we go for a walk?  
 B -> {}: a vase

Example 4.13: Breaks TA, respects QA, single addressee

his question to B. B gets the opportunity to answer the question, and then C answers the question. The test-phrase “I wasn’t asking you” is added in parentheses.

A -> {B}: What do you want for your birthday?  
 B -> {}: a vase  
 C -> {}: I want a car  
 (A -> {C}: I wasn’t asking you)

Example 4.14: respects TA, breaks QA, single addressee

Example 4.15 shows a dialogue in which both TA and QA are violated. A directs the question to B, but C answers the question before B. According to our hypothesis this example should be more awkward than Examples 4.13 and 4.14. In our opinion it is indeed impolite, but experiments are needed to confirm the hypothesis.

In Example 4.16 both QA and TA are respected. The question is directed to B and C, and D waits until both B and C have had the opportunity to speak before he introduces the next issue.

Example 4.17 shows a dialogue where QA is respected, because D doesn’t answer the question directed to B and C. TA is however broken, since someone other than D is selected as the next speaker.

Example 4.18 is somewhat unclear. TA is respected, since B gets the chance to answer A’s question before D introduces the new issue. It could be that C agrees completely with B, but normally, because of the manner in which the question is posed, an answer is expected from both B and C.

A -> {B}: What do you want for your birthday?  
 C -> {} a car  
 (A -> {C}: I wasn't asking you)  
 B -> {}: a vase

Example 4.15: breaks TA and QA, single addressee

A -> {B, C}: Are your watches really right? What do they show, exactly?  
 B -> {}: 13:47  
 C -> {} Mine says 13:45  
 D -> {B} nice watch! is it an Omega?

Example 4.16: respects TA and QA, multiple addressees

In Example 4.19, QA is violated, since D answers the question without being addressed. TA is respected. Maybe this kind of violation, where all the addressees have the chance to answer first (along with the one in Example 4.14) is considered milder than violations where this is not the case, as in Example 4.20.

Example 4.21 shows the ultimate rule breaking. A question is directed to B and C. D ignores the addressing completely and answers the question, thus violating turn assignment and violates QA towards all addressees.

If a DP *Q* asks *A*<sub>1</sub> and *A*<sub>2</sub> a collective question, he selects them as

1. respondents of the question, with a right to address the question, and with a common obligation to make sure that the question is answered
2. the set of DPs from which the next speaker should be selected. The actual selection can be thought of as a kind of self-selection.

Assume that *A*<sub>1</sub> starts speaking first. *A*<sub>1</sub> can now act in two ways, depending on if she thinks that the answer is complete. If the DP thinks so, the response will be directed to *Q*. If *A*<sub>2</sub> and *Q* disagree with *A*<sub>1</sub> regarding the completeness of the answer, they may take action to correct the answer. *Q* may select *A*<sub>2</sub> as the next speaker, or *A*<sub>2</sub> may self-select to offer an alternative or complementary answer.

If *A*<sub>1</sub> thinks that the answer is incomplete, *A*<sub>1</sub> will respond to the question in a way that selects *A*<sub>2</sub> as the next speaker – using a questioning tone, selecting *A*<sub>2</sub> with gaze or gesture, or by verbally selecting *A*<sub>2</sub> (“or what do you think...?”, “*A*<sub>2</sub>...?” or similar).

A -> {B, C}: Are your watches really right? What do they show, exactly?  
 D -> {B} nice watch! is it an Omega?

Example 4.17: breaks TA, respects QA, multiple addressees

A -> {B, C}: Are your watches really right? What do they show, exactly?  
 B -> {}: 13:47  
 D -> {B} nice watch! is it an Omega?

Example 4.18: respects TA & QA, multiple addressees

If  $A_2$  thinks that the answer that  $A_1$  has given is complete, or a valid partial answer to the question, she can signal this by agreeing with the utterance of  $A_1$ . If  $A_2$  disagrees with the utterance by  $A_1$ , she may signal this. As the collective obligation regards *making sure* that the question is answered, the obligation is resolved as soon as the question has been resolved.

If  $Q$  asks  $A_1$  and  $A_2$  a distributive question, he selects them as

1. respondents of the question, with a right to address the question, and with a distributed obligation to answer the question.
2. the set of DPs from which the next speaker should be selected. The actual selection can be thought of as a kind of self-selection.

The distributed obligation is an obligation for each of the individuals addressed to respond to the question. The interpretation of an agreement or a disagreement after an answer to a distributive question is questionable. The answer of  $A_1$ , resolving the question asked, does not resolve the obligation of  $A_2$  to answer the question.

We conclude by presenting relevant rights and obligations concerning turn and question management for the conversation roles in Table 4.2.

| Role   | TA rights                     | TA oblig   | QA rights   | QA oblig  |
|--|-------------------------------|--|---|---|
| Overhearer   | None                          | None   | None  | None  |
| Participant<br>(Addressee<br>and Side<br>Partici-<br>pant) | None                          | To not<br>take the<br>turn.  | To ground<br>ungrounded<br>material<br>on all lev-<br>els except<br>acceptance. | To follow<br>the conver-<br>sation.                       |
| Addressee<br>(collective)                                  | To make<br>one utter-<br>ance | To make<br>sure that<br>someone<br>in the<br>addressee-<br>set takes<br>the turn | To answer the<br>question, or<br>to accept/re-<br>ject an earlier<br>answer.    | To make<br>sure that<br>the ques-<br>tion is<br>answered. |
| Addressee<br>(distribu-<br>tive)                           | To make<br>one utter-<br>ance | To make<br>sure that<br>someone<br>in the<br>addressee-<br>set takes<br>the turn | To answer the<br>question.  | To an-<br>swer the<br>question.                           |

Table 4.2: Rights and obligations in connection to conversation roles.

A -> {B, C}: Are your watches really right? What do they show, exactly?  
 B -> {}: 13:47  
 C -> {}: Mine says 13:45  
 D -> {}: Mine is ten to two.  
 (A -> {D}): I wasn't asking you)

Example 4.19: respects TA, breaks QA, multiple addressees

A -> {B, C}: Are your watches really right? What do they show, exactly?  
 B -> {}: 13:47  
 D -> {}: Mine says ten to two.  
 (A -> {D}): I wasn't asking you)  
 C -> {}: Mine is 13:45

Example 4.20: respects TA, breaks QA, multiple addressees

## 4.4 Solving the Problems

### 4.4.1 AMA protocols

Some of the new protocols that are needed will be created by the AMA (Add Multiple Addressees) principle. We will use “reverse engineering” – we will first create the appropriate protocols, and then present the principle. The behaviour we are looking for is a more relaxed version of ADR, a blend of ADR and ASP, which allows, but does not require, every addressee to respond.

**The AMA Querying Protocol** When a speaker ( $A$ ) addresses a group of DPs ( $B, C_1, \dots, C_n$ ) with a collective question, one of the addressees ( $B$ ) will take the turn (by self-selecting) and make a max-qud-specific utterance. Then, each of the remaining addressees ( $C_1, \dots, C_n$ ) will also get an opportunity (not an obligation) to make a max-qud-specific utterance.

```

A -> {B, C}: Are your watches really right? What do they
               show, exactly?
D -> {}:      Mine is ten to two.
(A -> {D}):   I wasn't asking you)
B -> {}:      13:47
C -> {}:      Mine is 13:45

```

Example 4.21: breaks TA and QA, multiple addressees

```

LatestMove = Ask(A,{B, C1, ..., Cn},q)
A:   push q onto QUD; release turn
B:   push q onto QUD;
      take turn;
      make max-qud-specific utterance;
      release turn
Ci: push q onto QUD;
    (optional:
     take turn;
     make max-qud-specific utterance;
     release turn)

```

The ask dialogue move above is represented as a triple of the speaker, the addressee set and the question. The protocol guarantees that all the addressees follow what have been said, and that their respective QUD structures are identical. It also guarantees at least one max-qud-specific utterance in response to the query, and allows for all of the addressees to respond.

Note that the protocol above accommodates the process of answering not only collective questions, but also distributive questions. Answering Distributive questions are however better modelled using the original Duplicate Responder protocol.

**The AMA Assertion Protocol** If a group of speakers are addressed with an assertion, we would like to allow communal acceptance (see Section 4.3.3), but also give the opportunity for every addressee who wishes so to challenge the assertion. The protocol would look like this:

```

LatestMove = Assert(A,p)
A:      push p? onto QUD;
        release turn
B:      push p? onto QUD;
        take turn;
        (Option 1: Discuss p?,
         Option 2: Accept p)
Ci:     push p? onto QUD;
        (optional: take turn;
         (Option 1: Discuss p?,
          Option 2: Accept p)
        )
LatestMove = Accept(B,p)
B:      increment FACTS with p;
        pop p? from QUD;
Ci:     increment FACTS with p;
        pop p? from QUD;
A:      increment FACTS with p;
        pop p? from QUD;

```

The protocol guarantees that at least one of the addressees either accepts  $p$  or challenges  $?p$ . FACTS is not incremented until at least one party has made an acceptance move, but one acceptance move is enough to increment everybody's FACTS. More than one addressee may make an acceptance move.

**AMA principle** The multiple addressees basically duplicate the responder role, but without being obliged to say anything. Thus, the principle for creating the protocols will be as follows:

*Given a dialogue protocol  $\pi$ , add roles  $C_1, \dots, C_n$  which duplicate the responder role, but with the responder contributions optional.*

#### 4.4.2 The Agreement Move

When a collective question is directed to a group of DPs, and the first DP gives an answer to the question that the other DPs agree with, dialogues like the one in Example 4.22 happen. But what is the contribution of  $C$  here? Is it an acknowledgement? Yes, the utterance is clearly signalling to the other parties that  $C$  has understood and accepted the

DPs: {A, B, C, D, E}

A: Hey guys, do you have the time?  
 B: Yes, half past two.  
 C: Yeah  
 A: Thanks!

Example 4.22: Asking a group of people for the time, explicit agreement

DPs: {A, B, C, D, E}

A: Hey guys, do you have the time?  
 B: Yes, half past two.  
 A: Thanks!

Example 4.23: Asking a group of people for the time, no explicit agreement

utterance of B. Is it something else as well? Most likely. To see this, consider also the dialogue in Example 4.23. Imagine that it turns out after the dialogue that the hour of the day was very important to A, and that the time, when A asked, was instead half past four. To defend himself – to underline that he is innocent – C can then state that B is responsible for the error. He cannot do so after the dialogue in Example 4.22<sup>9</sup>. The move that he makes indicates that he is taking responsibility for the correctness of B’s report. This is something more than acknowledgement. The situation after his agreement is as if he had actually had uttered “It is half past two”.

We will model this as a kind of acceptance. After an assertion of  $P$  is made, an acceptance question  $P?$  will be pushed on QUD. A DP agreeing with  $P$  will then perform an agreement move. The agreement move will be interpreted as if the DP would have uttered the assertion of  $P$  himself. Disagreeing will be the opposite of agreeing. A disagreement move will be made by a DP disagreeing with  $P$ , and the interpretation of the move will be as if the disagreeing DP would have made an assertion of  $\neg P$ <sup>10</sup>.

We will follow the convention of (Larsson, 2002), and represent the acceptance and rejection moves as ICM (Interactive Communication Management) moves on the form `icm#(acc#Pol)` – where `Pol` is the polarity (positive or negative).

A correct treatment of the agreement move require some modifications to the handling of commitments. A suggestion of such a modification is discussed in Chapter 6.

---

<sup>9</sup>C could maybe be blamed for not correcting B’s error, given that C was paying attention to B and that he knew the correct time.

<sup>10</sup>Typically, disagreement will lead to discussion and argumentation, but this is not something that will be dealt with here.

### 4.4.3 Grounding

Larsson (2002) distinguishes three strategies for grounding – *pessimistic*, *optimistic* and *cautiously optimistic* grounding. Pessimistic grounding requires explicit evidence of contact, perception, understanding and acceptance before updating the DGB/IS. Optimistic grounding means updating DGB/IS without any evidence that the utterance has been understood etc. The cautiously optimistic strategy means applying the optimistic strategy, but with the opportunity to revise the DGB/IS in case of evidence for non-understanding etc.

The utterance processing protocols (both Ginzburg's and the ones presented in this thesis) implement a pessimistic strategy regarding acceptance. FACTS are not incremented until an acceptance move has been made. We will abandon that strategy in favour of a cautiously optimistic strategy in the spirit of Larsson (2002).

We also believe that the only functional grounding strategy for a side participant is the cautiously optimistic strategy. The reasons for this belief is the limited attention of the speaker reserved for the side participants – constantly monitoring the side participants for acknowledgements simply demands too much of the speakers attention<sup>11</sup>.

### 4.4.4 Obligations

David Traum argues in (Traum, 2003b) that Obligations and QUD are both necessary to model conversation. Traum tries to draw a clear line between the roles of QUD and obligations. QUD is seen as a short-term structure, primarily for ellipsis resolution. Obligations is seen as the long-term structure, keeping track of unresolved issues and who is to answer them. Compare this to Larsson's distinction between ISSUES and QUD (Section 2.3.2), where ISSUES can be regarded as a collection of “anonymous obligations”: An issue in ISSUES has been raised, but is still unresolved, and there is an implicit obligation that all the issues in ISSUES should be resolved. There are no obligations that are specific for the individual DP, but on all the DPs as a collective. Since a collective question can be considered resolved as soon as a resolving answer is received from any of the addressed DPs (see Section 4.3.4), we can draw the conclusion that the ISSUES structure is sufficient for modeling collective questions, given that such questions are not directed to a proper subset of the participants.

Our view is that an obligations layer *may* be motivated if the goal is to model other obligations as well – obligations not tied to dialogue behaviour – and if a single coherent

---

<sup>11</sup>Ginzburg and Fernández (2005b) use this argument as a motivation for the existance of “communal acceptance”.

framework to manage obligations is desired. However, it seems like the obligations in (Traum, 2003b; Traum and Rickel, 2002) are closely tied to the questions on QUD.

We will assume that the kind of obligations that we see in the scope of this thesis can be built into the logic of the agents and the QUD-processing update rules without compromising the behaviour or the design of the agents. We may need to introduce a few extensions to the objects stored on QUD to achieve what we want.

One question raised in (Traum, 2003b) is when one should see a question as resolved. (Ginzburg, 1996) suggests subjective goal-fulfilment as the criterion to be used, at least for two-party dialogue. We would like to argue that the risk that someone will reject an assertion is higher in a multi-party dialogue than in a two-party dialogue, and that we need a more cautious downdate condition. Therefore, we will add a condition on distance in turns between the last “q-specific” move and the QUD downdate. Each question will be left on QUD until there is no “q-specific” move in latest-moves. In this way, the other participants will be given a fair chance to react to an answer before the issue is removed from QUD. This is a cautiously optimistic downdate strategy.

#### 4.4.5 Extending QUD

If speaker  $A$  makes a query move with content  $Q$ ,  $Q$  should be pushed on the QUD stack. If other issues are stacked above  $Q$ , how should we remember who we should direct the response to? The obvious solution is to push the name of the querier of  $Q$ ,  $A$ , along with  $Q$  on the stack. QUD will therefore contain both the “owner” of the question and the question itself. We will do the same with the addressees of the question. If a query is directed to  $B$  and  $C$ , we will store the names of the recipients,  $B$  and  $C$ , along with in order to know who has the right to address the issue. We will henceforth refer to the set of addressees for a question  $Q$  as the  $ASET$  of  $Q$ .

We also need to monitor the fulfilment of obligations. We cannot use the  $ASET$ , because we need to modify the set in order to indicate obligation fulfilment. We will therefore introduce the  $OSET$  of  $Q$  as the set of addressees with an obligation either to answer  $Q$  or to make sure that  $Q$  is answered.  $OSET$  is, during the entire lifespan of a QUD-object, a subset of  $ASET$ . Initially,  $OSET$  will contain all the members of  $ASET$ . When  $Q$  is satisfactorily answered,  $OSET$  will be the empty set.

The element stacked on QUD will hence be a complex object, a quadruple,  $(S, Q, ASET, OSET)$  where  $S$  is the sender (speaker) of the question,  $Q$  is the question,  $ASET$  is the set of addressees and  $ASET$  is the set of DPs responsible (in some sense) to answer  $Q$ . If  $ASET$  is empty, the addressee will be determined by addressing-by-attribution. How should we decide what DPs should be in  $ASET$  (and  $OSET$ )? We will assume that the selection

techniques that are used for assigning the turn can be used for assigning the right and obligation to address a question as well.

## **4.5 Summary**

In this chapter, we designed a multi-party IBDM dialogue manager. After an initial review of the conversational roles defined by Clark & Carlson and Goffman respectively, and Fernández & Ginzburg's QUD protocols, based on the conversational roles, we argued for and suggested another set of protocols for multiple addressees, suggesting that the difference between some of the roles can be expressed in terms of grounding. We also introduced the distinction between collective and distributive questions, in order to account for differences regarding obligations and grounding. Finally, we suggested a modified turn manager to cope with multiple addressees, and a modified QUD, to be able to manage the rights and obligations to answer questions.

# Chapter 5

## Implementing a Multi-Party Dialogue Manager

To account for the data presented in examples in Chapter 4, in addition to a multi-party turn manager we also need a sequencing mechanism, implementing the features we have discussed in Section 4.4. We will base the implementation of the dialogue manager on the SCXML version of IBiS, presented in Chapter 2. The dialogue manager statechart of Multi-IBiS is depicted in Figure 5.1.

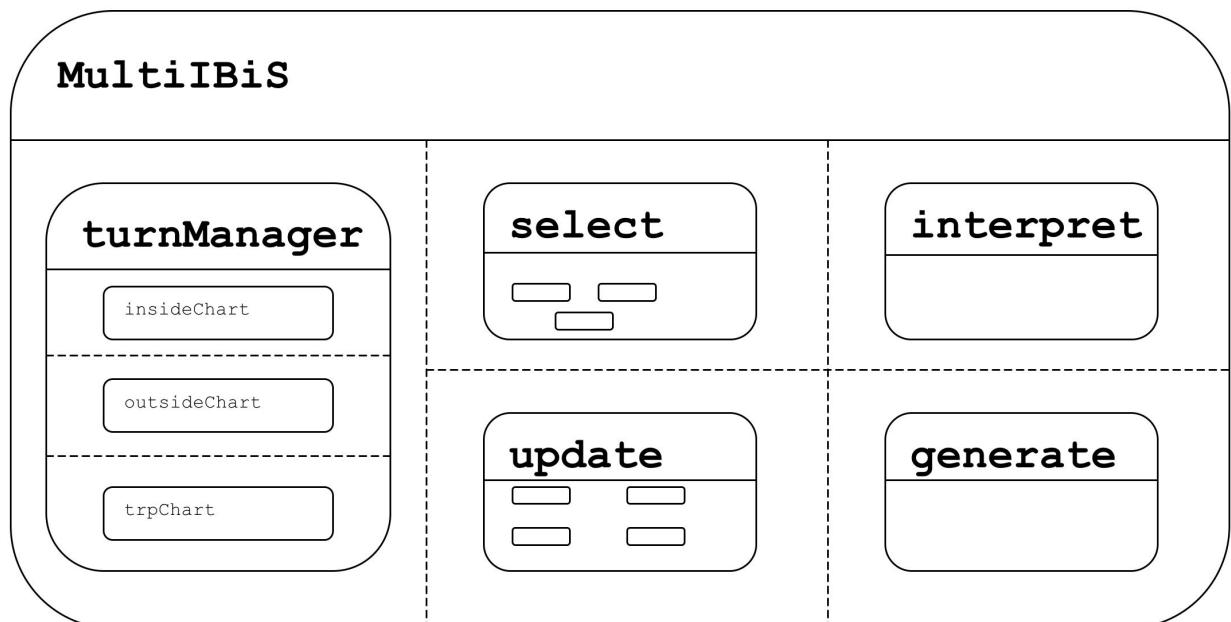


Figure 5.1: Multi-IBiS statechart

## 5.1 Modifications of the Turn Manager

The turn manager described in Chapter 3 is not sufficient to handle more than one addressee, unless every time more than one DP is selected as the next speaker, the DP equipped with our turn manager is selected first or not at all. Even if this would be the case, the machinery does not adequately describe what is happening.

To improve the turn manager, we need to introduce a set of states describing the situations that can occur when more than one DP is selected. We will call this new kind of TRP *shared TRP*.

A shared TRP is a TRP where the previous speaker has selected a set of speakers, from which the next speaker will be selected. This happens in the following way: A speaker starts speaking and then addresses one speaker and then one or more other speakers<sup>1</sup>. The sequence of events that is significant for a shared TRP is thus

[startSpeaking addressing addressing].

We also need to know who is addressed. We will allow ourselves to relax the requirements on the turn manager a bit. We will not distinguish a shared TRP where this agent is not included in the addressee set from an `othersTRP` (a TRP where another DP has the right and obligation to take the turn). So the sequence of addressing that is interesting for us is one where either someone else is addressed first and then this agent, or the other way around – a shared TRP where this agent is a member of the addressee set.

If we look at the TRP Chart of the current turn manager, we will see that the event sequence

[startSpeaking addressing]

will lead us to either `othersTRPComingUp` or `myTRPComingUp`, depending on who was addressed. The actual states representing the Shared TRP will be similar to the other ones: one state `sharedTRPComingUp` and one state `sharedTRP`. We want to reach the state `sharedTRPComingUp` when recognising the sequence mentioned above. Since we know that we are in either `othersTRPComingUp` or `myTRPComingUp` after seeing the first two events of the sequence, we add transitions from the two states to `sharedTRPComingUp`. The case where this agent is addressed first and then another one is taken care of by a transition from `myTRPComingUp` to `sharedTRPComingUp`, labelled `addressing(X) [X!=me]`. The other case is handled by a transition from `othersTRPComingUp` to `sharedTRPComingUp`, labelled identically.

---

<sup>1</sup>A speaker can obviously address more than one speaker at a time. We will however require such events to be transformed into a sequence of individual addressing events.

These are the major changes to the design of the turn manager. A few minor changes and additions (mainly event generation) are accounted for in Section 5.4. The new TRP Chart is shown in Figure 5.2.

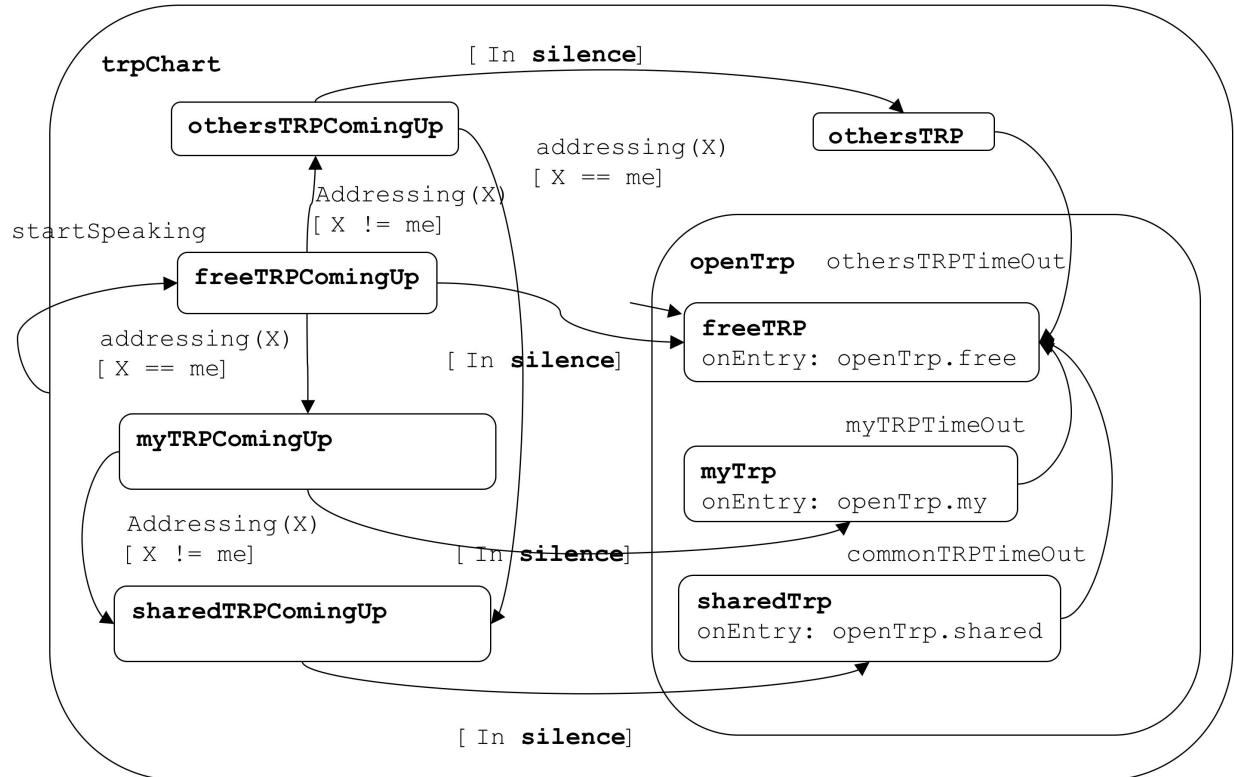


Figure 5.2: The TRP Chart extended with the shared TRP

## 5.2 The Information State

The datamodel of the multi-party enabled dialogue manager is shown in Listing 5.1. In the actual implementation, the datamodel includes a number of other things, for instance some class definitions etc. This printed version is cleaned up to increase readability. The datamodel contains

- the “Information State Proper” (Larsson, 2002), rooted in the element IS.
  - the name of the agent (**AgentName**)
  - the names of all the participating agents (in DPs)
  - and information about what DP is addressing what other DPs (in **Addressing**).

```

<datamodel>
  <data name="AgentName" expr="tSys"/>
  <data name="DPS" expr="[tSys]"/>
  <data name="Addressing" expr="{NewDictionary}"/>
  <data name="IS">
    <private>
      <agenda> {New Queue init()} </agenda>
      <plan>
        <issue> none </issue>
        <thePlan> {New Stack init()} </thePlan>
        <openForMe> false </openForMe>
      </plan>
      <bel> {New Set init()} </bel>
    </private>
    <shared>
      <com> {New Set init()} </com>
      <qud> {New StackSet init()} </qud>
      <lu>
        <moves> {New Set init()} </moves>
        <spkr> unit </spkr>
      </lu>
    </shared>
  </data>
</datamodel>

```

Listing 5.1: The multi-party Information State

### 5.2.1 Information State Proper

The Information State is, as usual, divided into two parts, `private` and `shared`, where `private` corresponds roughly to the UNPUB-MS of Ginzburg and `shared` to Ginzburg's DGB.

In this version, the `agenda` is implemented as a queue instead of the stack used in the previous version<sup>2</sup>. The `agenda` is the planning tool for the near future, and the queue allows us to handle the actions in chronological order. In a more advanced system it may be motivated to replace the queue with a priority queue to be able to override the chronological ordering in certain cases.

The `plan`, which is the long term planning storage, used to be a simple stack of actions

---

<sup>2</sup>The use of a stack in the original IBiS1 is only motivated by the fact that Larsson (2002) uses a stack as the default data structure.

to take to resolve a certain issue. In this implementation the `plan` has three daughter elements. The `IS.private.plan.thePlan` element holds the actual plan, while the `IS.private.plan.issue` stores the issue that this plan is intended to resolve. The `IS.private.plan.open` element contains a boolean value, representing information whether the issue is open for contributions from this agent. The `bel` element is unchanged, and holds a set of propositions.

In the `shared` element, there are the usual daughter elements – `com`, for storing the facts to which the DPs in the dialogue are committed, and a `StackSet`, `qud`, for storing questions under discussion. The contents of the QUD stack look a bit different from before, which we will see in Section 5.3.2. Finally there are the elements `lu` and `pu`, each containing a set `moves` holding the moves, and an element `spkr` holding the name of the speaker of the last utterance and the utterance before that, respectively. `IS.shared.lu` is updated by the integration rules. `IS.shared.pu` is updated as soon as an agent starts speaking.

### 5.2.2 Other Variables

The `AgentName` simply contains the identifier of this agent, to be used for identifying if dialogue moves, `startSpeaking`, `stopSpeaking` and `addressing` events are emanating from this agent or some other DP.

`DPs` is a list of the participating agents. It is updated with the name of a DP (if needed) as soon as a DP starts speaking, or if another participant addresses a DP. The initial value is the list containing only the identifier of this agent.

`Addressing` is a dictionary (similar to a hash-table), containing, for each of the DPs, a set of currently addressed DPs. The currently active (or open) QUD-objects keep a reference to this addressee-set and when an agent stops speaking, the reference from this dictionary to the set is thrown away. The set is updated as soon as an agent addresses another agent. If the set is empty when an agent stops speaking, the addressee set is assumed to be the set of all DPs.

### 5.2.3 Plans

In the plans (part of the domain knowledge), there has been an extension of the action elements constituting the plans. The `findout` and `respond` items in the plan have been extended with an agent name, so that it is known to whom this agent should address the utterances which are the result of the actions. Annotation of the plans, using the correct agent name, is done when loading the plan. This implementation does not support more

than one addressee for the moves generated by this agent.

## 5.3 The Dialogue Move Engine

The Dialogue Move Engine from Chapter 2 was a complex sequential state, where the update and selection modules, which were also represented as complex sequential states, were activated depending on who was the latest speaker – the user or the system. In practice, the transitions turn out to be very regular (select-update-update-select-update-update and so on). Of course, this is a consequence of the “normal” two-party dialogue structure.

The multi-party dialogue manager cannot expect this kind of regular transition pattern, because there is no guarantee for such a simple structure in multi-party dialogue. We have chosen a design where the update module is activated upon any dialogue move, and where the selection module is activated when it *seems* like the next TRP is going to be one where we have the right or obligation to speak. It is also activated at the event of any open TRP (a TRP where this agent has the right to speak). The selection module can be activated several times before making an utterance.

### 5.3.1 The Update Algorithm

The update module is represented as a complex, sequential state where each class of rules corresponds to one state. In order to avoid redundancy, the `init` and `grounding` states have been removed. The only task for the `init` state was to clear the agenda. In this implementation the need to clear the agenda before each `update` is removed. The `grounding` task was to move the content of the dialogue moves (the `says` events) into a field in the information state, where it could be explored by the integration rules. It is easier to let the integration rules react to the dialogue moves instead, and the reason for not doing this in the previous version was to provide a translation as close to the original as possible.

In the update module, there are the following states (or rule classes):

`integrate`, `downdateQUD`, `loadPlan` and `execPlan`. The `update` state of Multi-IBiS is shown in Figure 5.3.

`integrate` is the initial state of the `update` state. When an event named `says` occurs in the event queue, a transition of the `integrate` class applies. In case where none of the standard integration rules apply, a default rule is applied. The default rule logs the information state (for debugging reasons) and adds “notMeaningful” to the set of latest

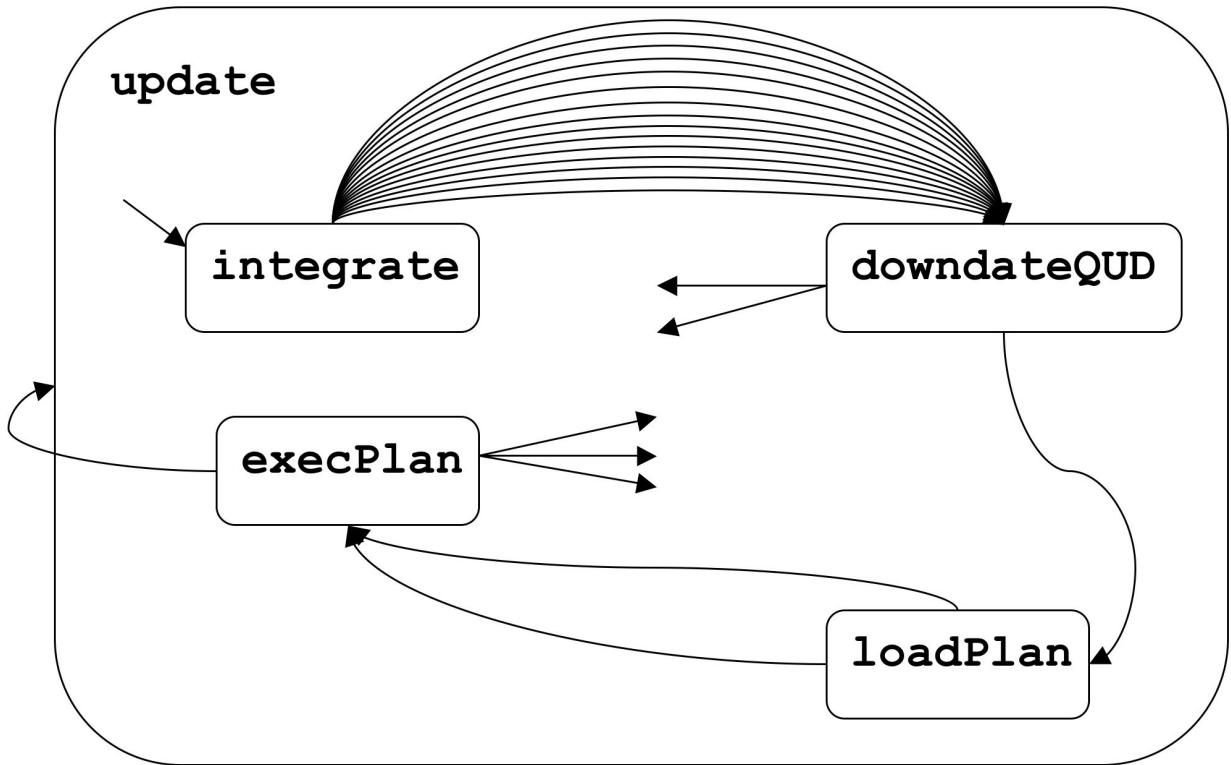


Figure 5.3: The Update State of MultiIBiS

moves.

Only one of the `integrate` transitions should trigger per `says` event, which is guaranteed since all of the transitions reside inside the same state, and all the transitions have the state `downdateQUD` as their target. If one transition applies, the state will be left and no other transitions with source state `integrate` will trigger.

### 5.3.2 The QUD object

The items stacked on QUD are, as we described in Section 4.4.5, no longer simple questions. Instead, we create a kind of QUD-object. We implement the QUD-object as an Oz dictionary. The entries of the dictionary is described in Table 5.1.

| Field               | Description   |
|---------------------|---|
| <code>issue</code>  | The question that this object describes.  |
| <code>spkr</code>   | The DP raising the issue.   |
| <code>ASET</code>   | The set of DPs with the right to address this issue.  |
| <code>OSET</code>   | The set of DPs with obligations regarding providing an answer to this question.   |
| <code>status</code> | the addressing status of this object, <code>open</code> or <code>closed</code> . Signals if the ASET/OSET of this object can be extended by addressing (if the utterance where this issue was introduced is finished or not). |

Table 5.1: The fields of the QUD object.

### 5.3.3 Individual, Collective and Distributive questions

The update rules will require different operations for collective and distributive questions. We will not solve the problem of identifying collective and distributive questions. We will assume that certain questions are always collective while the others are always distributive, ignoring the contextual factors.

### 5.3.4 The Update Rules

**Removed Rules** The transition `getLatestMove` is removed, simply because there are no moves to “get”. The integrate transitions are triggered directly by the `says` events.

There are no transitions or rules for quitting the system. The agents are supposed to be running constantly, so a more suitable type of move would be a dialogue closing move. This is however not implemented. There are no rules for greetings.

**integrateOtherAsk** The transition `integrateUsrAsk` has a new name, `integrateOtherAsk`, since the transition will apply every time a question is asked by some other DP than this agent, and that DP isn’t necessarily a user.

In the preconditions of the transition, we check that the move type is “ask” and we check that the speaker name is not equal to this agent’s name (in Chapter 4 we used the hard-coded names `usr` and `sys`). We also check that there is a plan available, which we can execute in order to obtain an answer to the question. The QUD-object is then pushed on QUD, and the action to respond to the question is added (enqueued) on the agenda.

**integrateOtherAskNoPlan** The standard `integrateOtherAsk` transition requires the existance of a plan that can provide the agent with a sequence of actions to perform in order to come up with an answer to the question.

This agent needs to integrate also questions for which there is no such plan. One reason is that an agent, according to the principle of responsibility, has an obligation to follow the conversation and therefore needs to integrate also questions which it cannot answer. Another reason is that the agent may have certain beliefs that makes it possible for the agent to answer the question, even without a plan.

The transition is very similar to `integrateOtherAsk`. The condition that a suitable plan should be available is removed. Instead a condition that the question should have a relevant answer is added. If we cannot understand the question, and judge what is a relevant answer to the question, it cannot be integrated.

The action to add a `respond` item to the agenda is also removed. In the case where the question asked is an individual or distributive question, it is replaced with an action to reject the issue (if this agent has the information to answer the question the rejection action is eventually removed from the agenda).

**integrateMeAsk** Even if the name `integrateSysAsk` is still valid in some sense, we decided to change its name into `integrateMeAsk` to keep the correspondence to `integrateOtherAsk`.

The transition is very similar to `integrateOtherAsk`, with the exception that answering the question should not be a goal for this agent – no action is added to the agenda.

**integrateAnswer** The rule `integrateAnswer` has become a family of transitions for integration of answers. The standard answer integration transition is `integrateAnswer`. The conditions for the transition are unchanged from the two-party version:

- The current move should be an `answer` move.
- The content of the current move is `A`.
- There should be an object with `issue Q` on `QUD` to which `A` is a relevant answer.

The actions executed if the preconditions apply are the following:

- The negation of  $P$  – the proposition which is the result of combining  $Q$  and  $A$  – should be removed from the commitments.
- $P$  should be added to the commitments.
- The “old” QUD-Object,  $Q$ , should be pushed on QUD, in order to reflect the fact that the issue is discussed (pushing an element which is already a member of the StackSet means that it is moved to the top of the StackSet).
- If the issue of  $Q$  is a collective question, the OSET of  $Q$  is cleared. Otherwise, the OSET is decremented with the speaker of the answer-move.
- A new QUD-Object for the acceptance question  $\text{accept}(P)$  should be constructed and pushed on QUD. The object’s issue field should be  $\text{accept}(P)$ . We assume that the ASET of this acceptance-question is the ASET of  $Q$  with the addition of the DP who raised  $Q$ , given that the issue is a collective question. If the question is distributive or individual, the only member of ASET will be the DP who raised the issue.

The commitments strategy is a variety of cautiously optimistic grounding of (Larsson, 2002). Since the data structure used for the commitments is an ordinary set, we must explicitly delete propositions in conflict with  $P$ . In our simple environment the only potential proposition in conflict with  $P$  is  $\text{neg}(P)$ .

The transition `integrateAnswerAgree` handles integration of answers to which this agent agrees. The additions to the preconditions is that this agent actually agrees with  $P$  – this agent should believe that  $P$  – and that the question to which this move is an answer is a collective question. If we do so, we add the action to *agree* with  $P$  to the agenda. Since it is already clear that the question is collective, we clear the OSET of the issue.

The opposite to agreement is disagreement. The transition for cases where this agents believes that  $\text{neg}(P)$  is `integrateAnswerDisagree`. The transition is identical to `integrateAnswerAgree` with the following exceptions: This agent should believe that  $\text{neg}(P)$  (instead of  $P$ ), and the action added to the agenda is to *disagree* with  $P$ . It may be surprising that  $P$  is added to commitments even if the agent plans to disagree with  $P$  at a later stage. The opportunity to disagree with  $P$  may however not come, and since we assume (cautiously) optimistic acceptance of answers, and since commitments are part of a shared structure, the agent need to commit to  $P$  until we have rejected  $P$  to keep the commitments consistent.

As we (mostly) stick to the acceptance model formulated by Ginzburg (using acceptance questions), we also need to generate acceptance and rejection moves – moves that signal whether an assertion by a DP is accepted by its addressee(s) or not. For this purpose we have implemented the transitions `integrateAnswerAccept` and `integrateAnswerReject`.

The basic conditions and actions of these transitions are very similar to the ones of `integrateAnswer`. We condition the acceptance and rejection moves on this agent being speaker of the question to which the current move is an answer and that this agent does not believe that `neg(P)` (acceptance), or otherwise believes that `neg(P)` (rejection).

We do not take addressing into account in the integration transitions. Even if this agent is not addressed, the action to agree with, disagree with, accept or reject P is added to the agenda. It is not until the selection phase that addressing information is checked, since we do not know if we will be part of the addressee-set until the speaker has stopped speaking.

**Integration of Acceptance and Rejection** Since we have introduced the possibility to produce acceptance and rejection moves, it is reasonable that we can integrate them as well.

We will use ICM moves, described in (Larsson, 2002), to represent the acceptance and rejection moves. Acceptance is handled differently from (Larsson, 2002), but the moves for accepting and rejecting propositions are nevertheless relevant to us.

The form of an ICM move is `icm:ActionLevel*Polarity`, where `ActionLevel` is one of `con` (contact), `per` (perception), `sem` (semantic understanding), `und` (pragmatic understanding) or `acc` (acceptance/reaction), and `Polarity` is either `pos`, `neg` or `interrogative`. An ICM move may also carry some content – in our case a proposition `P` or an issue `issue(Q)`.

Since acceptance is the only ICM move we will consider, the set of ICM moves will consist of only two items, positive and negative acceptance. In our notation they will look like the following: `{icm#(acc#pos) icm#(acc#neg)}`. A positive acceptance move will typically be realised as “yes”, “okay”, “yes, I think that P” or similar (given a positive proposition as the object for acceptance). A rejection move can be realised as “no”, or “I don’t think that P” or similar (given the same restrictions). The `says` event communicating the ICM move can optionally carry a proposition (in the case of answer acceptance) or a question (in the case of issue rejection).

**Acceptance and Rejection of answers** The first transition under this heading is `integratePosAcceptICM`. For an acceptance move not containing a proposition to be meaningful, it is necessary that the topmost issue on QUD is an acceptance question. Because of the optimistic commitment strategy employed in this system, the action part of the transition is empty.

The preconditions for integration of rejection moves are identical to the integration of

acceptance moves. The actions are however different: rejection of an assertion (`answer`) means that the commitments need to be updated – the proposition  $P$ , which is the object for the acceptance question, should be removed. Also the obligations connected to the question  $Q$  to which  $P$  is an answer needs to be updated – we add the identifier of the agent responsible for the rejected move to the `OSET` of the QUD-object of  $Q$ .

We have also added the possibility of long-distance rejection and acceptance in case the acceptance issue is not QUD-maximal. The transition covering long-distance rejection is called `integrateLDNegAcceptICM`, the transition for acceptance is called `integrateLDPoSAcceptICM`. The preconditions state that the issue `accept(P)` must be a member of `IS.shared.qud` (viewing QUD as a set). To identify the correct acceptance-issue, there is also a requirement that the proposition  $P$  should be included in the `says` event as `contents`. The action parts are identical to the action parts of `integrateNegAcceptICM` and `integrateLDPoSAcceptICM` respectively, with the addition of pushing the issue `accept(P)` on QUD. Since QUD is a stack-set, the operation of pushing a member  $M$  of QUD on QUD has the effect of moving  $M$  to the top of QUD, making it available for acceptance/rejection moves in the next turn. Before pushing the QUD-object of `accept(P)`, we push the QUD-object of the underlying question on QUD.

**Issue Rejection** Integration of issue rejection is handled by the transition `integrateNegIssueAcceptICM`. The preconditions state that the contents of the move should be `issue(Q)`, where  $Q$  is an issue on QUD, and the speaker of the move must be a member of the `OSET`. The effects of the transition is that the speaker is deleted from the `OSET` of the issue, and that  $Q$  is re-pushed on QUD, making it the topmost QUD-object.

**downdateQUD** In the two-party variant of IBiS1, there is one single transition in the `downdateQUD` class. We have extended the class into two transitions, `downdateQUD` and `downdateQUDAcceptance`.

The transition `downdateQUD` has become more complex, since we have changed the nature and function of QUD. In the previous version, with QUD implemented as a “pure” stack, the topmost element on QUD was the only accessible element and therefore the one used to interpret any answer. QUD was downdated as soon as an answer matching the topmost element was found.

In the current implementation, QUD cannot be downdated after any resolving answer, since different DPs may give other answers to issues. We push acceptance issues on top of QUD. These issues cannot be downdated on the basis of resolvedness only.

A non-acceptance issue  $Q$  will be removed from QUD if

- the obligations to answer  $Q$  have been resolved –  $\text{OSET}$  is empty
- there is no proposition  $P$ , which is the result of combining  $Q$  with a relevant answer, in this agent's beliefs, which this agent has not committed to (this agent has no unexpressed beliefs regarding  $Q$ )
- and there are no recent moves relevant to  $Q$ . Relevant moves are answers to  $Q$  and acceptance moves.

An acceptance issue  $\text{accept}(P)$  will be removed from QUD if there are no recent relevant moves. Relevant moves are acceptance and rejection moves which carry the contents  $P$  and, in the case of an acceptance issue topmost on QUD, no contents at all. An answer  $A$  is relevant to an acceptance issue  $\text{accept}(P)$  if there is a question  $Q$  which can be combined with  $A$  to form  $P$ . The preconditions of  $\text{downdateQUD}$  are shown in Listing 5.2

```

{IS.shared.qud member(Q0)}
{Q0.oset isEmpty()}
Q0.issue = Q
{Lib.nonAcceptance Q}
{IS.shared.lu.moves nonMember(move(ask Q))}
{IS.shared.lu.moves nonMember(move(icm#(acc#neg)
                                issue(Q)))}
for A in
  {Search.base.all proc {$ Ans}
    {Domain.relevantAnswer Q Ans}
    end
  } do
  {IS.shared.lu.moves nonMember(move(answer A))}
end

for P in {IS.private.bel toList($)} do
  BelQ
in
  {Lib.combine BelQ _ P}
  if BelQ == Q then
    {IS.shared.com member(P)}
  end
end

```

Listing 5.2: The preconditions part of  $\text{downdateQUD}$

**findPlan** The transition `findPlan` has been updated in two ways. The first change has to do with the inclusion of agent names in some plan items (see Section 5.2.3). The second change is caused by the need to make sure that a question has been assigned to this agent before we start executing the plan. We have extended the information state element `IS.private.plan` to include three subelements, `issue` – the issue which the plan resolves, `openForMe` – whether this agent may address that issue, and `thePlan`, for the plan itself. The first step in the selection process, before copying items from the plan to the agenda, is to check that we can be certain that this agent may address the issue. If so, the flag `openForMe` is set to true.

**removeFindout** The transition `removeFindout` is functionally equivalent to the two-party version.

**execConsultDB** The only changes to `execConsultDB` has to do with the inclusion of agent names in some items (see Section 5.2.3).

### 5.3.5 The Selection “Module”

The selection module is built up from two major states, or subcharts, run in parallel: `selectNoreply` and `selectNormal`. Selection charts are idle until they are activated by the event `doSelect`. The logic of the state `selectNormal` tries to find a move to make, and if this does not happen (i.e. the event `generate.*` does not happen within a specified amount of time) when this agent has the right and obligation to speak (`{In myTrp}`), `selectNoreply` generates a move with the contents that the agent knows that it is expected to say something, but that no suitable move was found.

**selectNormal** The normal selection process happens in two steps, corresponding to the two rule classes in the original IBiS1, `selectAction` and `selectMove`. The former, is responsible for dispensing actions from the plan to the agenda, the later realises the items on the agenda into actual moves and sends them away for generation, using the event `generate.generateMove`.

**selectRespond** The rule `selectRespond` is responsible for finding questions to which this agent has an answer, and to add an action to respond the question. The preconditions are that the question should be open for this agent to respond, that this agent should know of an answer to the question, and that the participants should not already be committed

to that answer. The action part of the transition adds to the agenda the action to respond to the question, addressing the asker of the question.

**selectAnswer** In the previous version of the dialogue manager, the preconditions of **selectAnswer** were that the first element of the agenda should be a **respond(Q)** item, that the proposition, which is the result of combining  $Q$  with the relevant answer  $R$ , should be in this agent's beliefs. In this multi-party version, while the abovementioned preconditions still remain (although slightly modified), we also need to add a number of new preconditions.

The modifications are the following: The **respond** item on the agenda has a second argument,  $DP$ .  $DP$  designates the  $DP$  to address with the **answer** move.

In order not to answer a question directed to someone else, it must be checked that there is an issue  $Q$  on QUD, for which the ASET contains the identifier of this agent. We check that the proposition  $P$ , the combination of  $Q$  and  $R$ , is not a member of the shared commitments, in order for the agent not to state the obvious.

The action part of this transition consists of sending an event carrying information about the move type, the contents of the move and about the addressee of the move. The event name is **generate.generateMove**. We also dequeue the **respond** item from the agenda.

**selectPosAcceptanceICM** This transition is responsible for generating positive acceptance moves addressing an **accept** issue in QUD. The preconditions require that the first element in the agenda is an **accept** item. It is also required that there is an issue in QUD, **accept(P)**, which has an ASET status which allows this agent to address it.

The action part of the transition specifies that the **accept** element on the agenda should be dequeued, and that the event **generate.generate** should be emitted. The data payload contains information about the move type (**icm#(acc#pos)**), the contents ( $P$ ) and addressing information (that  $DP$  should be the addressee of the move).

**selectNegAcceptanceICM** The transition for selecting negative acceptance moves works the same way as the transition for selecting positive acceptance moves, with the expected differences.

The transitions requires the item **reject** instead of **accept** to be the first element in the agenda. The move to be generated is of the type **icm#(acc#neg)** instead of **icm#(acc#pos)**.

If none of the acceptance transitions are carried out, the `reject` or the `accept` item on the agenda is removed by a transition, `dropAcceptance`.

**selectNegIssueAcceptanceICM** The mechanism for selecting a negative acceptance move to reject an issue involves three transitions – two for removing the rejection element from the agenda without selecting the move, and one for actually selecting the move.

The rejection item is introduced every time that a question  $Q$ , for which we do not have any corresponding plan, is introduced by some other DP than this agent. To answer a question, we need to have a belief  $P$ , which is a proposition resulting from combining  $Q$  with an answer  $A$ . If we do have this belief  $P$ , we should not reject the issue  $Q$ .

The first transition takes care of that case, checking that there is a rejection item in the agenda, that there is an issue  $Q$  on QUD which we can answer. The result of applying the transition is that the rejection item is dequeued from the agenda.

The second transition takes care of the cases where we should actually reject an issue. The conditions for application is that there is a rejection item, `reject(Q DP,` on the agenda, that there is an issue  $Q$  on the agenda, and that this agent is a member of the OSET of  $Q$ . The result of applying the transition is that an move `icm#(acc#neg)` with contents `issue(Q)`, directed to DP is sent for generation.

In other cases, the rejection should be dropped. Of course, it would be better to have on single rule taking care of all cases, but such a rule would be rather complicated given the limitations of the interface to the datamodel objects.

**selectAsk** The transition in itself is almost as simple as the two-party version. If the first item in the agenda is a `findout` element, an `ask` move is generated. To prevent that more than one question is asked in the same utterance (an arbitrary choice that seem to make sense, inspired by (Larsson, 2002)), there is a parallel state surveiling the moves that has been selected. As soon as a question has been selected, the state `noQuestions` is exited. Therefore we have added the condition that this state should be active in order for the dialogue manager to be able to select an `ask` move.

To prevent that a question  $Q$ , not open for this agent to answer, are addressed with other questions in the process of executing a plan to answer  $Q$ , there is a machinery (in `selectAction`) that delays execution of a plan until we know for sure that we may address the question  $Q$ .

## 5.4 Integrating Turn and Dialogue management

We will now merge the turn manager designed in Chapter 3 into the multi-party dialogue manager. The major changes to the turn manager have already been described in Section 5.1 above. This section will explain how the code in the dialogue manager and the turn manager needs to be changed in order for the two to function together.

### 5.4.1 Assigning the Right to Address an Issue

The QUD objects contain, as we said earlier, fields for information about the issue, the speaker (who introduced the issue), and for addressing information. The addressing information consists of the *ASET* and *OSET* of the issue and information about if the addressing information can be updated or not. This information needs to come from somewhere.

**Simplifying Assumptions** The assignment of issues are relatively uncertain – remembering Clark’s words about a vocative that could have been used in an utterance signalling that at least one of the illocutionary acts of that utterance being directed to that party. We will however assume that all of the moves in an utterance are directed to the same set of DPs.

We assume that the signals for selecting the next speaker are identical to the signals used for assigning the right to address a question. We do not know if this is a fact or not, but for the reason of simplicity we will assume this as a fact.

Another assumption is that the addressing set cannot be updated once the speaker has stopped speaking. This is possibly not true (cf. example in (Traum, 2003b, p. 11), where a question is passed on to a DP, who was not in the addressee set from the beginning). A possible solution is to let addressing events which are not occurring in connection with a dialogue move increment the addressee set of the topmost unresolved QUD-object.

**Question assignment events** This simplification makes it possible to use the same addressing events as the turn manager uses for deciding on turn assignment issues.

We use a simple state, parallel to the other states, to integrate the addressing events. Each time an addressing event occurs, we iterate through the items on QUD, checking that the name of the speaker responsible for the addressing event is equal to the name in the `spkr`

field of the QUD object, and that the field status is set to `open`. If the conditions are met, the set in the `addr` field incremented with the new addressee.

The field `status` is set to `open` on creation of the object. We have chosen to remove the possibility to update the addressing information once the speaker has ceased speaking. Each time a `stopSpeaking` event is received, we iterate through the QUD items to check if the name in the `spkr` field is equal to the `speaker` field of the event data payload and if `status` is set to `open`. In that case `status` is changed into `closed`.

### 5.4.2 Delegating Selection Control to Turn Manager

**A Word on Feedback** In Section 3.4.4 we described how to take care of negative feedback adequately. When this agent’s talk is overlapped by negative feedback, an event is emitted. The event is received in the generation state, where a transition stops the current output.

Positive feedback has, according to (Duncan, 1972), the function of avoiding the turn. In our turn manager, positive feedback has the effect of resetting the turn manager – every occurrence of a `startSpeaking` event makes the `freeTrpComingUp` active, all addressing information is forgotten. This means that even unaddressed DPs who give positive feedback reset the turn manager. However, the question assignment information is not forgotten, and our guess is that the two systems together form a unit robust enough to handle even these situations. Of course, this has to be evaluated in a live system.

### Selection Control in IBiS1

In the two party version of IBiS1, the user and the system take turns when speaking. Turn is regulated indirectly from the control algorithm (see Listing 5.3). First, a move is selected by the system, which is then given a “surface form” and realised (by the `generate` and `output` modules). After the system have carried out its moves and updated the IS accordingly, it starts listening to the user and gives the utterance an interpretation (`input` and `interpret` modules) after which the IS is again updated. This is then repeated until the user or the system executes a “quit” move and the execution of the program is stopped.

In the multi-party version, we cannot expect this kind of “one for me – one for you” behaviour. Instead the turns are managed by events which are interpreted by the turn manager which transduces the sequences of such signals to TRP events (discussed in Chapter 3).

```

repeat(
  select
  if not is empty($next_moves) then(
    generate
    output
    update
  )
  test($program state == run)
  input
  interpret
  update
)

```

Listing 5.3: The control algorithm of the original IBiS1

Since the turn manager is designed to emit events when TRPs open to this agent are present, it is reasonable that this information is used to control the activities of the **Select** state.

## Selection Control in Multi-IBiS

When a TRP, which is open to this agent, is approaching, an event `openTRP.comingUp.(my|shared|free)` is emitted from the turn manager. When the TRP is present, an event `openTRP.here.(my|shared|free)` is emitted. Both types of events trigger a transition in the generation state, which in turn emits an event `doSelect`. This event triggers the selection of moves in the selection state (see Section 5.3.5).

The selected moves are communicated to the generation state using `generate.generateMove` events. The moves are added to the `QueueSet`<sup>3</sup> `NextMoves`. As soon as there is at least one move in `NextMoves` to be performed, and if the state `OpenTRP` in the turn manager is active, the generator will start speaking. Selected moves (selection triggered by a `openTRP.*.*` event) which are not generated and output (the TRP turned out to be closed, or someone else started speaking before us) are discarded, in order to avoid situations where the moves are outdated when uttered (QUD may look different, questions may already be answered etc.).

---

<sup>3</sup>A `QueueSet` is a queue which enforces the set property on a queue, in order to avoid the same move being added more than once.

### 5.4.3 Unused Features of the Turn Manager

There is a feature in the turn manager of emitting the event `overlapResolved` when an overlap has been resolved. The dialogue manager described in this chapter does not make use of this event. But if one is to build a conversational agent which is to be deployed in a game setting, it could be preferable to equip the agent with mechanisms for holding the turn, when the agent's right to the turn is challenged (see (Schegloff, 2000) for descriptions of human strategies to hold the turn in an overlap situation). Information about when the right to the floor has been secured will be needed by such an agent to help deciding when the fight is won (or lost).

## 5.5 Tracing Multi-IBiS

### 5.5.1 Agreement and Disagreement

We give traces from two scripted runs of multi-Ibis. The first trace demonstrates agreement. The multi-Ibis agent `tSys` has the belief that the destination is London (destination is a collective question in this example). The agent `a` asks about the destination, speaker `b` responds with “London”. The move is integrated using the transition `integrateAnswerAgree`, adding the action to agree with the answer to the agenda. Finally, `tSys` agrees, and the agreement is integrated using the transition `integratePosAcceptICM`. For the full trace, see Appendix B.

```

01030    Event:      startSpeaking      o(speaker:a)
01240    Event:      utterance
               o(contents:'What is the city of destination' speaker:a)
01700    Event:      integrated
               o(contents:destination move:ask speaker:a)
01710    Log:       TransTaken:      integrateOtherAskNoPlan
01850    Event:      stopSpeaking     o(speaker:a)
02020    Event:      startSpeaking    o(speaker:b)
02170    Log:       IS.private.bel   [ destination # london ]
02210    Log:       max-qud
               o( aset:[ ] issue:destination oset:[ ] spkr:a status:open )
02230    Event:      utterance      o(contents:'London.' speaker:b)
02840    Event:      stopSpeaking    o(speaker:b)
03080    Event:      integrated
               o(contents:london move:answer speaker:b)
03090    Log:       TransTaken:      integrateAnswerAgree
03650    Log:       TransTaken:      selectPosAcceptanceICM

```

```

04210 Log:      IS.shared.com      [ destination # london ]
04220 Log:      max-qud
  o( asset:[ a tSys ]
    issue:accept( destination # london )
    issueObject: o( asset:[ tSys ]
      issue:destination
      oset:[ ]
      spkr:a
      status:closed )
  oset:[ a tSys ]
  spkr:b
  status:closed )
04560 Event:    startSpeaking   o(speaker:tSys)
04690 Event:    utterance       o(contents:'Yes, London.' speaker:tSys)
04720 Event:    addressing      o(addressee:b speaker:tSys)
05820 Log:      IS.private.bel [ destination # london ]
05860 Log:      IS.shared.com  [ destination # london ]

```

The second trace, below, demonstrates disagreement. At the start of the conversation **tSys**, the Multi-IBiS agent, has the beliefs that the destination is not London, but Paris. DP **a** asks about the destination and **b** answers “London”. The utterace is integrated by **tSys** by making the transition **integrateAnswerDisagree**, and as a result the action to disagree with **destination#london** is added to the agenda. When selecting moves, the **selectRespond** transition is carried out, adding the action to respond to the question about destination. The transition is triggered by the fact that the question about destination is open to **tSys**, and that there is a proposition in the beliefs of **tSys** resolving the question. The **selectRespond** transition adds the action to respond to the question to the agenda. Both moves are directed to the asker of the question. For the full trace, see Appendix C.

```

01240 Event:    utterance
  o(contents:'What is the city of destination' speaker:a)
02170 Log:      IS.private.bel
  [ destination # paris neg( destination # london ) ]
02220 Event:    utterance   o(contents:'London.' speaker:b)
03100 Log:      TransTaken: integrateAnswerDisagree
03570 Log:      TransTaken: selectRespond
03660 Log:      TransTaken: selectNegAnswerAcceptanceICM
03720 Log:      TransTaken: selectAnswer
04850 Event:    startSpeaking o(speaker:tSys)
05200 Event:    utterance
  o(contents:'No, not London.' speaker:tSys)
05280 Event:    addressing    o(addressee:b speaker:tSys)
05880 Log:      TransTaken: integrateNegAcceptICM

```

```

06420  Event:      utterance
        o(contents:'I am going to Paris.' speaker:tSys)
06450  Event:      addressing      o(addressee:a speaker:tSys)
07370  Log:       TransTaken:    integrateAnswer
07700  Event:      stopSpeaking   o(speaker:tSys)
07850  Log:       IS.shared.com  [ destination # paris ]

```

### 5.5.2 Acceptance and Rejection

`tSys` uses an optimistic strategy for acceptance. As soon as an answer is given (at 3220 ms), `tSys` adds the resulting proposition to the commitments. When the rejection comes at 5230 ms, the commitment is removed, and as soon as the new answer is uttered the new one is added. The acceptance at 9250 has no effects. For the full trace, see Appendix E.

```

01250  Event:      utterance
        o(contents:'What is the city of destination' speaker:a)
02260  Log:       IS.shared.com  [ ]
03220  Event:      utterance      o(contents:'London.' speaker:b)
03730  Event:      integrated
        o(contents:London move:answer speaker:b)
03730  Log:       TransTaken:    integrateAnswer
04260  Log:       IS.shared.com  [ destination # london ]
05230  Event:      utterance      o(contents:'No.' speaker:a)
05700  Event:      integrated
        o(contents:unit move:icm#(acc#neg) speaker:a)
05710  Log:       TransTaken:    integrateNegAcceptICM
06320  Log:       IS.shared.com  [ ]
07240  Event:      utterance      o(contents:'Paris.' speaker:a)
07750  Event:      integrated
        o(contents:Paris move:answer speaker:a)
07760  Log:       TransTaken:    integrateAnswer
08280  Log:       IS.shared.com  [ destination # paris ]
09250  Event:      utterance      o(contents:'Yes.' speaker:b)
10310  Log:       IS.shared.com  [ destination # paris ]

```

### 5.5.3 Question Assignment

The dialogue is started by `a` asking `b` a question about travel prices, which is what `tSys` is specialized in. There is no answer from `b`. The question is then repeated at 07230, addressing no-one in particular. This time, `tSys` considers the question to be open for its

contributions, and starts executing the plan for finding out the answer to the question. For the full trace, see Appendix D.

```

01240  Event:      utterance
        o(contents:'Travel price information please.' speaker:a)
01350  Event:      addressing      o(addressee:b speaker:a)
01700  Event:      integrated
        o(contents:travelPrice move:ask speaker:a)
01710  Log:        TransTaken:    integrateOtherAsk
01860  Event:      stopSpeaking   o(speaker:a)
02240  Log:        IS.private.plan.openForMe   uncertain
05660  Log:        plan:       notForMe
07230  Event:      utterance
        o(contents:'Travel price information please.' speaker:a)
07870  Event:      stopSpeaking   o(speaker:a)
08470  Log:        plan:       indeedForMe
09340  Event:      utterance
        o(contents:'How do you want to travel?' speaker:tSys)

```

#### 5.5.4 Information Sharing by Overhearing

The example shows how information is shared by overhearing. The scripted agents **a** and **hSys** are discussing the hotel price issue, which is outside the domain of **tSys**. However, there is a question (at 02640 ms) and an answer (at 03720 ms) which **tSys** understands (integrates) – the proposition that the destination is London is added to the commitments. When **a** later asks about travel prices (at 06100 ms), the issue about destination is never raised, but removed from the plan at 19230 ms. For the full trace of the first 30 seconds of this scripted interaction, see Appendix F.

```

01260  Event:      utterance
        o(contents:'Hotel price information please.' speaker:usr)
01820  Log:       TransTaken:      defaultIntegrate
02640  Event:      utterance
        o(contents:'What is the city of destination' speaker:hSys)
03190  Event:      integrated
        o(contents:destination move:ask speaker:hSys)
03200  Log:       TransTaken:      integrateOtherAskNoPlan
03720  Event:      utterance      o(contents:'London.' speaker:usr)
04460  Event:      utterance
        o(contents:'The hotel price is 1234 Euro' speaker:hSys)
04640  Event:      integrated
        o(contents:london move:answer speaker:usr)
04650  Log:       TransTaken:      integrateAnswer
05610  Log:       IS.shared.com      [ destination # london ]
06100  Event:      utterance
        o(contents:'Travel price information please.' speaker:usr)
06160  Event:      integrated
        o(contents:hotelPrice#1234 move:answer speaker:hSys)
06170  Log:       TransTaken:      defaultIntegrate
07800  Event:      integrated
        o(contents:travelPrice move:ask speaker:usr)
07810  Log:       TransTaken:      integrateOtherAsk
08020  Log:       TransTaken:      findPlan
10700  Event:      utterance
        o(contents:'How do you want to travel?' speaker:tSys)
12350  Event:      utterance
        o(contents:'By train.' speaker:usr)
15150  Event:      utterance
        o(contents:'Yes, by train.' speaker:tSys)
16310  Event:      utterance
        o(contents:'What is the city of departure?' speaker:tSys)
18350  Event:      utterance      o(contents:'Paris.' speaker:usr)
19230  Log:       TransTaken:      removeFindout
19240  Log:       Removed:       findout(destination usr)

```

## 5.6 Summary

In this chapter, we have implemented a multi-party dialogue manager, based on the Issue-based dialogue management model and on the findings in Chapter 4. We have extended the turn manager from Chapter 3 with a mechanism for selecting the next speaker from a set of DPs. We extended the QUD-object (the type of item stored in QUD) with entries

for obligation and rights management, and added the concept of collective and distributive questions to the model. We then implemented the update rules to reflect the utterance processing protocols from Chapter 4, adapted to the new structure of the QUD-objects and to the suggested question types. We integrated the turn manager into the dialogue manager, giving it the role of a controller for the move-selection process. Finally we gave examples of traces from running the dialogue manager in simulated dialogues.



# Chapter 6

## Conclusions and Future Research

### 6.1 Summary

In this thesis we have described implementations in SCXML of various aspects of dialogue management with a focus on multi-party dialogues.

In Chapter 1 we described some of the problems in connection to multi-party dialogue – dialogue management and turn management – and tried to solve them using a rather naïve approach. The approach proved unfruitful, the resulting dialogue managers were not flexible enough. The formalism used for describing the dialogue managers was Harel statecharts.

In Chapter 2 we described the ISU and IBDM approaches to dialogue management, and proposed a manner to implement a simple IBDM dialogue manager – IBiS1 – in SCXML (which is an XML rendering of Harel statecharts).

In Chapter 3 we built a turn manager, using the SSJ model as a blueprint. The turn manager is built from three main components, each responsible for one well-defined subtask of turn management.

Chapter 4 was dedicated to multi-party dialogue management. We reviewed relevant previous work and argued that some of that work needed to be modified and extended in order to properly describe multi-party dialogue. We introduced a distinction between distributive and collective questions in order to account for differences regarding how the questions are answered. Finally, we extended the objects stored on QUD to include also information about the speaker and the addressees of the question.

The implementation of the multi-party dialogue manager MultiIBiS was the theme of Chapter 5. We extended the IBDM dialogue manager from Chapter 2 to handle also multi-party dialogue, using the extensions from Chapter 4. It was also integrated with the turn manager from Chapter 3.

## 6.2 Concluding discussion

### 6.2.1 Dialogue Management

We have built a dialogue manager which handles multi-party dialogue. It is not a complete dialogue system, ready to hook up on the Internet or to connect to an incoming telephone line, but nevertheless it is an important proof-of-concept – IBDM and the QUD-related theories on dialogue really scale up to multi-party dialogue. The dialogue manager handles individual, collective and distributive questions. It can generate and interpret agreement and disagreement, and also acceptance and rejection. It can identify and handle occurrences of shared TRPs, where the current speaker does not select an individual to become the next speaker, but instead selects a set of participants of which the next speaker should be a member.

The dialogue manager only partially models the shared obligation to answer a question when more than one DP is addressed: if a number of DPs are addressed with a question, and none of the addressees answers the question, this agent will not take action. We are not certain about how strong the obligation to reject a collective question is. There seems to exist a repair strategy for situations like this, where the DP asking the question confronts the non-answering DPs with the fact that she has asked a question and expects an answer, alternatively wants an explanation to why the question is not answered.

In Section 4.2.3 we quoted a question from (Ginzburg and Fernández, 2005b): “should the contents be considered grounded when any of the addresses has acknowledged them? Should evidence of understanding be required from every addressee?” We believe that we can answer that question now. If we assume a cautiously optimistic strategy for grounding and for QUD downdate, together with the AMA protocol for assertion, we can answer the question in the following way:

The contents of an utterance can be considered grounded when all the addressees have been given a fair chance to report negative acknowledgement and all side participants have been given a fair chance to report negative acknowledgement regarding contact, perception and understanding, and none of them has done so.

If we assume a pessimistic grounding strategy, the contents of an utterance can be considered grounded when all of the addressees have acknowledged them and all side participants have been given a fair chance to report negative acknowledgement regarding contact, perception and understanding, and none of them has done so.

To some extent, our account of questions in multi-party dialogue relies on identifying collective and distributive questions. In this thesis we have used a simplification, looking up question types in a table. Exploring how the two types of questions can be identified in context is a subject for further research – crucial for the usefulness of some of the models put forward in this thesis.

### 6.2.2 SCXML

SCXML is not yet a standard – the only publications so far have been drafts. We do not know what SCXML will look like when the standard is set. However, the state-machine parts of the specification have been relatively stable, while the datamodel access seems incomplete. This means that the basic design of the statecharts we have described in this thesis will most likely remain largely untouched, while the code for datamodel access will most likely need to be redesigned.

The SCXML implementation that we have used for the documents described in this thesis is also relatively immature. The objects integrated with the datamodel is more or less an ad-hoc solution, and the blend of function-like and query-like methods of the objects make the design of the transitions a challenge. Much code needs to be duplicated, due to the fact that the prolog-like queries are not available in the `cond` attribute of the `<if>` element.

We have seen some performance issues when running the XML documents we have developed in this thesis. We are not certain about the cause for these issues – a possible cause is inefficient formulations of the transition conditions. This must however be investigated before a massive migration to SCXML. Dialogue systems are sensitive to timing, and even small lags may have a negative impact on the usability.

We want to highlight what we think are the strong points of SCXML (also put forward in (Kronlid and Lager, 2007)). SCXML is:

- Intuitive. Statecharts and thus SCXML are based on the very intuitive yet highly abstract notions of states and events.

- Expressive. It is reasonable to view SCXML as a multi-paradigm programming language, built around a declarative DFA core, and extended to handle also imperative, event-based and concurrent programming.
- Extensible. SCXML is designed with extensibility in mind (Barnett et al., 2006a), and our investigations suggest that there is indeed room for simple extensions that will increase the expressivity of SCXML considerably.
- Theory neutral. Although it is clear that the framework is suitable for implementing both simple DFA-based as well as form-based dialogue management, the framework as such is fairly theory neutral.
- Visual. Just like ordinary finite-state automata, statecharts have a graphical notation – for “tapping the potential of high bandwidth spatial intelligence, as opposed to lexical intelligence used with textual information” (Samek, 2002).
- Methodologically sound. The importance of support for refinement and clustering should not be underestimated. In addition, the fact that SCXML is closely aligned to statechart theory and UML will help those using model driven development methodologies.
- XML enabled. Thus, documents may be validated with respect to a DTD or an XML Schema, and there are plenty of powerful and user friendly editors to support the authoring of such documents.
- Part of a bigger picture. SCXML is designed to be part of a framework not just for building spoken dialogue systems, but also for controlling telephony – a framework in which technologies for voice recognition, voice-based web pages, touch-tone control, capture of phone call audio, outbound calling (i.e. initiate a call to another phone) all come together.
- Endorsed by the W3C. The fact that SCXML is endorsed by the W3C may translate to better support in tooling, number of implementations and various runtime environments.

In our experience, SCXML is an excellent language for describing and solving the problems related to dialogue management. The event model makes integration of components responsible for separate parts of dialogue management very easy. The developer only needs to make sure that the correct event is generated at the right time and that the event is interpreted in the correct way. Parallelism and synchronisation is already taken care of by the SCXML interpreter. Adding a debugging facility, such as printing the information state on the occurrence of a given event, is just a matter of creating an event handler – a transition listening to that certain event, logging the desired information. Simulating a

user or two to drive a dialogue is as easy as defining transitions which are listening to the relevant dialogue move and which are then responding with an appropriate utterance.

An SCXML statechart disregards events for which no transition is defined. If an active state is the source state for one single transition  $T$  listening only for the event  $e$ , and the event  $f$  is first in the event queue, nothing happens – except that  $f$  is thrown away. If  $f$  is of no significance in a certain state, the developer do not need to care about it. The advantage of this behaviour is that the code can be kept very clean and that the statechart becomes very robust. The downside is that important events may be forgotten if they occur when they are not expected, as no error message is generated when an event is dropped.

Despite the fact that SCXML was not (as far as we know) designed for the purpose of implementing the ISU approach to dialogue management, it is still possible to use it as such, provided the proposed rather moderate extension is made. Indeed, we believe that SCXML could potentially replace other ISU implementations such as TrindiKit and DIPPER – which should be good news for academic researchers in the field, as well as for the industry. Good news for researchers since they will get access to an infrastructure of plug-and-play platforms and modules once such platforms and modules have been built (assuming they *will* be built), good news for industry since a lot of academic research suddenly becomes very relevant, and good news for the field as a whole since SCXML appears to be able to help to bridge the gap between academia and industry.

The way we have used the SCXML datamodel, populating it with live Oz objects, is probably not what the designers of SCXML had in mind. A more elegant solution would be to have an all-XML object model, which allows objects to be stored as XML in the datamodel, and where the interface to the objects is expressed in XML. This interfacing XML (method calls etc.) should then be possible to use as children to `<transition>` elements, and elsewhere where executable content is allowed. SCXML is, as previously mentioned, designed to be extended, and an ISU-XML language to plug into SCXML is probably what is needed to provide the desired functionality.

### 6.2.3 Corpus Studies

In Chapter 4 we made use of a large number of constructed examples. Many of the examples illustrate occurrences of turn assignment and question assignment to more than one DP. Finding corpus evidence of such parallel assignments would strengthen the claims put forward in this thesis considerably. Particularly uses of the check-phrase “I wasn’t asking you” (or similar), indicating a violation of question assignment, would be valuable. We suspect, however, that this way of shutting down a dialogue participant is relatively rare and considered to be rather impolite – more impolite than answering a question intended

for someone else. Thus, most cases of question assignment violation are not marked.

### 6.2.4 QUD and ISSUES

Ginzburg (1996) defines QUD as a partially ordered set. QUD-maximal elements are available for elliptical answers, and no, one or several elements may be QUD-maximal at a given point in time. In Larsson's IBiS1, QUD is represented as a stack, and QUD-maximality corresponds to being the topmost element of the QUD stack. In consequence, one or no element may be QUD-maximal at a given point in time.

In Chapter 4, we modified QUD and the QUD downdate principle, so that also issues which are resolved remain on QUD for at least one turn. We have not defined QUD-maximality in our model, but allow any question in QUD to be answered using both non-elliptical and elliptical answers. This is not optimal.

Most likely, a solution with two structures like (Larsson, 2002), would be preferable, where ISSUES contains the QUD-objects for unresolved issues, and where QUD contains QUD-objects for recently discussed issues. The QUD-objects in the ISSUES structure would then correspond to unresolved obligations, and the objects in QUD would correspond to QUD-maximal questions, in some sense. Such a model will probably also simplify the implementation of rules for question accommodation and for action-oriented dialogue in MultiIBiS.

### 6.2.5 Semantics

In this thesis we have used a very simple semantics model, where propositions are represented as (possibly negated) pairs of a question and an answer. Most practical dialogue systems seem to do with relatively restricted semantic models but a real multi party dialogue system would most likely need a more elaborate semantics. A reasonable starting point would be the semantics of (Larsson, 2002), where (for instance) yes/no-questions and alternative questions are available, along with a more reasonable model for lambda expressions.

### 6.2.6 Individual Commitments

In the dialogue managers presented in this thesis, we have modelled commitments as propositions that all the participants in the conversation have committed to. This is a rather

blunt representation of commitment. For instance, the dialogue examples in 4.4.2 feature some things that are not captured in the “standard” representation of commitments.

Consider the dialogues in Examples 4.22 and 4.23, p. 95. The difference between the dialogues is that in the first dialogue,  $C$  explicitly has committed to the proposition “The current time is 14:30”, but not in the second one. The difference is however not recorded in the information state, since the commitments are collective – there is no trace of the individual commitment of  $C$ . To capture this aspect of the agreement move, introducing individualised commitments in the information state seems like the obvious choice.

The solution that we propose to be explored, is not trying to model the mental situations of the DPs, but to simply keep a record about who has explicitly committed to what in a dialogue. It will rarely be the case that there is one individual commitment  $P$  and another individual commitment  $\neg P$  – since we cannot say that there exists such commitments, except in cases where the DPs have explicitly “agreed to disagree”. The value of this kind of representation of individual commitments is uncertain in a two-party dialogue.

After the dialogue that we refer to above (Example 4.22, p. 95), such a model would show something like the following:

```
is.shared.com.a      []
is.shared.com.b      [current_time(14:30)]
is.shared.com.c      [current_time(14:30)]
is.shared.com.com    [current_time(14:30)]
```

This should be compared to the model after (Example 4.23, p. 95):

```
is.shared.com.a      []
is.shared.com.b      [current_time(14:30)]
is.shared.com.c      []
is.shared.com.com    [current_time(14:30)]
```

An agent deploying the individual commitments model that we propose can, in the later case, correctly integrate an utterance from  $C$  where he blames  $B$  for the faulty time report. Individual commitments may have other uses as well – for instance finding the right agents to address when challenging a commitment.

Individual commitments would also be useful to model cases of disagreement between the DPs, where they explicitly have ‘agreed to disagree’. Such disagreement cannot be modelled with the collective commitments used in this thesis and elsewhere.



# Bibliography

- Keith Allan. Hearers, overhearers, and Clark and Carlson's informative analysis. *Language*, pages 19–32, 1986.
- Jens Allwood. Obligations and Options in Dialogue. *Think*, 3:9–18, May 1994.
- Jens Allwood. Reasons for management in dialog. In R. J. Beun, M Baker, and M Reiner, editors, *Dialogue and Instruction*, pages 241–250. Springer-Verlag, 1995.
- Jim Barnett, RJ Auburn, Michael Bodell, Marc Helbing, Rafah Hosn, and Klaus Reifenrath (eds). State Chart XML (SCXML): State Machine Notation for Control Abstraction. Web document, <http://www.w3.org/TR/2006/WD-sxml-20060124/>, January 2006a.
- Jim Barnett, Michael Bodell, Dan Burnett, Jerry Carter, and Rafah Hosn (eds). State Chart XML (SCXML): State Machine Notation for Control Abstraction. Web document, <http://www.w3.org/TR/2007/WD-sxml-20070221/>, February 2006b.
- Johan Bos, Ewan Klein, Oliver Lemon, and Tetsushi Oka. DIPPER: Description and Formalisation of an Information-State Update Dialogue System Architecture. In *4th SIGdial Workshop on Discourse and Dialogue*, Sapporo, 2003. ACL.
- Justine Cassell, Timothy W. Bickmore, Hannes Hoge Vilhjalmsson, and H. Yan. More Than Just a Pretty Face: Affordances of Embodiment. In *Intelligent User Interfaces*, pages 52–59, 2000.
- Herbert H. Clark. What Is Said to Whom: A Rejoinder to Allan. *Language*, 62(3):518–529, September 1986.
- Herbert H. Clark and Thomas B. Carlson. Hearers and Speech Acts. *Language*, 58(2):332–373, 1982.
- Starkey Duncan, Jr. Some signals and rules for taking speaking turns in conversations. *Journal of Personality and Social Psychology*, 23:289–292, 1972.
- Jens Edlund, Mattias Heldner, and Joakim Gustafson. Utterance segmentation and turn-taking in spoken dialogue systems. *Computer Studies in Language and Speech*, 8:576–587, 2005.

- Arash Eshghi and Patrick G. T. Healey. Collective States of Understanding. In S. Keizer, H. Bunt, and T. Paek, editors, *Proceedings of the 8th SIGdial Workshop on Discourse and Dialogue*, pages 2–9. Association for Computational Linguistics, 2007.
- Brendan S. Gillon. Collectivity and distributivity internal to english noun phrases. *Language Sciences*, 18(1-2):442–468, 1996.
- Jonathan Ginzburg. Personal communication, 2007.
- Jonathan Ginzburg. *Semantics and Conversation*. CSLI Publications, 2008. Forthcoming.
- Jonathan Ginzburg. Interrogatives: Questions, facts, and dialogue. In Shalom Lappin, editor, *Handbook of Contemporary Semantic Theory*. Blackwell, Oxford, 1996.
- Jonathan Ginzburg and Raquel Fernández. Action at a distance: the difference between dialogue and multilogue. In *Proceedings of Dialor '05*, LORIA, Nancy, 2005a.
- Jonathan Ginzburg and Raquel Fernández. Scaling up to multilogue: some benchmarks and principles. In *Proceedings of the 43rd meeting of the Association for Computational Linguistics*, Ann Arbor, Michigan, 2005b.
- Erving Goffman. *Forms of Talk*. University of Pennsylvania Press, 1981.
- David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.
- Thomas K. Harris, Satanjeev Banerjee, Alex Rudnicky, June Sison, Kerry Bodine, and Alan Black. A research platform for multi-agent dialogue dynamics. In *Proc. of the IEEE International Workshop on Robotics and Human Interactive Communication*, pages 497–502, 2004.
- Joris Hulstijn and Gerard A. W. Vreeswijk. Turntaking: a case for agent-based programming. Technical report, Institute of Information and Computing Sciences, Utrecht University, 2003.
- Fredrik Kronlid. Turn Taking for Artificial Conversational Agents. In M. Klusch, T. Rovatsos, and T. R. Payne, editors, *Cooperative Information Agents X*, pages 81–95. Springer-Verlag, 2006.
- Fredrik Kronlid and Torbjörn Lager. Implementing the Information-State Update Approach to Dialogue Management in a Slightly Extended SCXML. In Ron Artstein and Laure Vieu, editors, *Proceedings of the 11th International Workshop on the Semantics and Pragmatics of Dialogue (DECALOG)*, pages 99–106, May 2007.
- Staffan Larsson. *Issue-Based Dialogue Management*. PhD thesis, University of Gothenburg, 2002.

- Staffan Larsson and David Traum. Information state and dialogue management in the TRINDI Dialogue Move Engine Toolkit. *Natural Language Engineering*, 6(3-4):323–340, 2000.
- David L. Martin, Adam J. Cheyer, and Douglas B. Moran. The Open Agent Architecture: A Framework for Building Distributed Software Systems. *Applied Artificial Intelligence*, 13(1-2):91–128, January-March 1999. OAA.
- Colin Matheson, Massimo Poesio, and David Traum. Modelling grounding and discourse obligations using update rules. In *Proceedings of the 1st Annual Meeting of the North American Association for Computational Linguistics (NAACL2000)*, May 2000.
- Emiliano G. Padilha. *Modelling Turn Taking in a Simulation of a Small Group Discussion*. PhD thesis, University of Edinburgh, 2006.
- Emiliano G. Padilha and Jean Carletta. A simulation of small group discussion. In *Proceedings of Edilog*, 2002.
- Harvey Sacks, Emanuel A. Schegloff, and Gail Jefferson. A simplest systematics for the organization of turn-taking for conversation. *Language*, 50:696–735, 1974.
- Miro Samek. *Practical Statecharts in C/C++*. CMPBooks, 2002.
- Emanuel A. Schegloff. Overlapping talk and the organization of turn-taking for conversation. *Language in Society*, 29(1):1–63, 2000.
- Kristinn R. Thórisson. Natural turn-taking needs no manual: Computational theory and model, from perception to action. In B. Granström, D. House, and I. Karlsson, editors, *Multimodality in Language and Speech Systems*, volume 19 of *Text, Speech and Language Technology*. Kluwer Academic Publishers, Dordrecht, 2002.
- David Traum. Issues in Multiparty Dialogues. In *Proceedings of International Workshop on Agent Communication Languages, ACL 2003*, pages 201–211. Springer Berlin / Heidelberg, July 2003a.
- David Traum. Semantics and Pragmatics of Questions and Answers for Dialogue Agents. In *Proceedings of the International Workshop on Computational Semantics*, pages 380–394, January 2003b.
- David Traum and Jeff Rickel. Embodied agents for multi-party dialogue in immersive virtual worlds. In *Proceedings of the first International Joint conference on Autonomous Agents and Multiagent systems*, pages 766–773, Bologna, Italy, July 2002.



# Appendix A

## Trace of IBiS1 run

```
00820 Trans: top:<cxm1> => IBiS1:<p>[dialogueMoveEngine select generate driver monitor]
00840 Trans: selectAction:<s> => selectMove:<s>
00920 Event: generateMove o(move:greet)
00920 Log: transition applied: selectOther
00920 Trans: selectMove:<s> => update:<s>
00940 Log: transition applied: init
00940 Trans: init:<s> => grounding:<s>
00950 Event: says o(move:greet speaker:sys)
01020 Log: transition applied: getLatestMove
01030 Event: says o(move:ask(price) speaker:usr)
01030 Event: says o(move:greet speaker:usr)
01040 Log: sys greet
01040 Trans: grounding:<s> => [says] => integrate:<s>
01060 Log: transition applied: integrateGreet
01060 Trans: integrate:<s> => downdateQUD:<s>
01080 Trans: downdateQUD:<s> => loadPlan:<s>
01090 Trans: loadPlan:<s> => execPlan:<s>
01160 Trans: execPlan:<s> => init:<s>
01170 Log: transition applied: init
01170 Trans: init:<s> => grounding:<s>
01240 Log: transition applied: getLatestMove
01280 Log: usr greet
01280 Trans: grounding:<s> => [says] => integrate:<s>
01300 Log: transition applied: integrateGreet
01300 Trans: integrate:<s> => downdateQUD:<s>
01320 Trans: downdateQUD:<s> => loadPlan:<s>
01330 Trans: loadPlan:<s> => execPlan:<s>
01360 Trans: execPlan:<s> => select:<s>
01410 Trans: selectAction:<s> => selectMove:<s>
01450 Trans: selectMove:<s> => update:<s>
01470 Log: transition applied: init
01470 Trans: init:<s> => grounding:<s>
01570 Log: transition applied: getLatestMove
01580 Log: usr ask(price)
01580 Trans: grounding:<s> => [says] => integrate:<s>
01680 Log: transition applied: integrateUsrAsk
01680 Trans: integrate:<s> => downdateQUD:<s>
01690 Trans: downdateQUD:<s> => loadPlan:<s>
01720 Log: transition applied: loadPlanT
01720 Trans: loadPlan:<s> => execPlan:<s>
01750 Trans: execPlan:<s> => select:<s>
01810 Log: transition applied: selectAction
01810 Trans: selectAction:<s> => selectMove:<s>
01840 Event: generateMove o(move:ask(how))
01850 Log: transition applied: selectAsk
01850 Trans: selectMove:<s> => update:<s>
01860 Log: transition applied: init
01860 Trans: init:<s> => grounding:<s>
01880 Event: says o(move:ask(how) speaker:sys)
01950 Log: transition applied: getLatestMove
01960 Event: says o(move:answer(train) speaker:usr)
01970 Log: sys ask(how)
01970 Trans: grounding:<s> => [says] => integrate:<s>
02020 Log: transition applied: integrateSysAsk
02020 Trans: integrate:<s> => downdateQUD:<s>
```

```

02080 Trans: downdateQUD:<s> => loadPlan:<s>
02090 Trans: loadPlan:<s> => execPlan:<s>
02130 Trans: execPlan:<s> => init:<s>
02140 Log: transition applied: init
02140 Trans: init:<s> => grounding:<s>
02250 Log: transition applied: getLatestMove
02250 Log: usr answer(train)
02250 Trans: grounding:<s> => [says] => integrate:<s>
02370 Log: transition applied: integrateAnswer
02370 Trans: integrate:<s> => downdateQUD:<s>
02390 Log: transition applied: downdateQUD
02390 Trans: downdateQUD:<s> => loadPlan:<s>
02400 Trans: loadPlan:<s> => execPlan:<s>
02480 Log: transition applied: removeFindout
02510 Trans: execPlan:<s> => select:<s>
02530 Log: transition applied: selectAction
02530 Trans: selectAction:<s> => selectMove:<s>
02590 Event: generateMove o(move:ask('from'))
02600 Log: transition applied: selectAsk
02600 Trans: selectMove:<s> => update:<s>
02620 Log: transition applied: init
02620 Trans: init:<s> => grounding:<s>
02630 Event: says o(move:ask('from') speaker:sys)
02670 Log: transition applied: getLatestMove
02680 Event: says o(move:answer(paris) speaker:usr)
02730 Log: sys ask('from')
02730 Trans: grounding:<s> => [says] => integrate:<s>
02780 Log: transition applied: integrateSysAsk
02780 Trans: integrate:<s> => downdateQUD:<s>
02800 Trans: downdateQUD:<s> => loadPlan:<s>
02810 Trans: loadPlan:<s> => execPlan:<s>
02880 Trans: execPlan:<s> => init:<s>
02900 Log: transition applied: init
02900 Trans: init:<s> => grounding:<s>
03000 Log: transition applied: getLatestMove
03010 Log: usr answer(paris)
03010 Trans: grounding:<s> => [says] => integrate:<s>
03120 Log: transition applied: integrateAnswer
03120 Trans: integrate:<s> => downdateQUD:<s>
03150 Log: transition applied: downdateQUD
03150 Trans: downdateQUD:<s> => loadPlan:<s>
03160 Trans: loadPlan:<s> => execPlan:<s>
03200 Log: transition applied: removeFindout
03260 Trans: execPlan:<s> => select:<s>
03280 Log: transition applied: selectAction
03280 Trans: selectAction:<s> => selectMove:<s>
03320 Event: generateMove o(move:ask(to))
03320 Log: transition applied: selectAsk
03320 Trans: selectMove:<s> => update:<s>
03380 Log: transition applied: init
03380 Trans: init:<s> => grounding:<s>
03390 Event: says o(move:ask(to) speaker:sys)
03440 Log: transition applied: getLatestMove
03450 Event: says o(move:answer(london) speaker:usr)
03460 Log: sys ask(to)
03460 Trans: grounding:<s> => [says] => integrate:<s>
03550 Log: transition applied: integrateSysAsk
03550 Trans: integrate:<s> => downdateQUD:<s>
03560 Trans: downdateQUD:<s> => loadPlan:<s>
03580 Trans: loadPlan:<s> => execPlan:<s>
03650 Trans: execPlan:<s> => init:<s>
03660 Log: transition applied: init
03660 Trans: init:<s> => grounding:<s>
03770 Log: transition applied: getLatestMove
03770 Log: usr answer(london)
03770 Trans: grounding:<s> => [says] => integrate:<s>
03850 Log: transition applied: integrateAnswer
03850 Trans: integrate:<s> => downdateQUD:<s>
03910 Log: transition applied: downdateQUD
03910 Trans: downdateQUD:<s> => loadPlan:<s>
03930 Trans: loadPlan:<s> => execPlan:<s>
03960 Log: transition applied: removeFindout
04030 Trans: execPlan:<s> => select:<s>
04050 Log: transition applied: selectAction
04050 Trans: selectAction:<s> => selectMove:<s>
04080 Event: generateMove o(move:ask(month))
04090 Log: transition applied: selectAsk
04090 Trans: selectMove:<s> => update:<s>
04100 Log: transition applied: init
04100 Trans: init:<s> => grounding:<s>
04110 Event: says o(move:ask(month) speaker:sys)
04210 Event: says o(move:answer(april) speaker:usr)
04210 Log: transition applied: getLatestMove
04220 Log: sys ask(month)
04220 Trans: grounding:<s> => [says] => integrate:<s>
04310 Log: transition applied: integrateSysAsk
04310 Trans: integrate:<s> => downdateQUD:<s>
04330 Trans: downdateQUD:<s> => loadPlan:<s>

```

```

04340 Trans: loadPlan:<s> => execPlan:<s>
04410 Trans: execPlan:<s> => init:<s>
04430 Log: transition applied: init
04430 Trans: init:<s> => grounding:<s>
04490 Log: transition applied: getLatestMove
04500 Log: usr answer(april)
04500 Trans: grounding:<s> => [says] => integrate:<s>
04610 Log: transition applied: integrateAnswer
04610 Trans: integrate:<s> => downdateQUD:<s>
04670 Trans: downdateQUD:<s> => loadPlan:<s>
04680 Log: transition applied: downdateQUD
04690 Trans: loadPlan:<s> => execPlan:<s>
04730 Log: transition applied: removeFindout
04760 Trans: execPlan:<s> => select:<s>
04810 Log: transition applied: selectAction
04810 Trans: selectAction:<s> => selectMove:<s>
04850 Event: generateMove o(move:ask('class'))
04850 Log: transition applied: selectAsk
04850 Trans: selectMove:<s> => update:<s>
04870 Log: transition applied: init
04870 Trans: init:<s> => grounding:<s>
04880 Event: says o(move:ask('class') speaker:sys)
04980 Event: says o(move:answer(business) speaker:usr)
04980 Log: transition applied: getLatestMove
04990 Log: sys ask('class')
04990 Trans: grounding:<s> => [says] => integrate:<s>
05080 Log: transition applied: integrateSysAsk
05080 Trans: integrate:<s> => downdateQUD:<s>
05100 Trans: downdateQUD:<s> => loadPlan:<s>
05110 Trans: loadPlan:<s> => execPlan:<s>
05150 Trans: execPlan:<s> => init:<s>
05160 Log: transition applied: init
05160 Trans: init:<s> => grounding:<s>
05260 Log: transition applied: getLatestMove
05270 Log: usr answer(business)
05270 Trans: grounding:<s> => [says] => integrate:<s>
05390 Log: transition applied: integrateAnswer
05390 Trans: integrate:<s> => downdateQUD:<s>
05410 Log: transition applied: downdateQUD
05410 Trans: downdateQUD:<s> => loadPlan:<s>
05460 Trans: loadPlan:<s> => execPlan:<s>
05500 Log: transition applied: removeFindout
05520 Log: transition applied: execConsultDB
05550 Trans: execPlan:<s> => select:<s>
05610 Log: transition applied: selectAction
05610 Trans: selectAction:<s> => selectMove:<s>
05630 Event: generateMove o(move:answer(1234))
05640 Log: transition applied: selectAnswer
05640 Trans: selectMove:<s> => update:<s>
05650 Log: transition applied: init
05650 Trans: init:<s> => grounding:<s>
05660 Event: says o(move:answer(1234) speaker:sys)
05770 Event: says o(move:quit speaker:usr)
05770 Log: transition applied: getLatestMove
05780 Log: sys answer(1234)
05780 Trans: grounding:<s> => [says] => integrate:<s>
05900 Log: transition applied: integrateAnswer
05900 Trans: integrate:<s> => downdateQUD:<s>
05920 Log: transition applied: downdateQUD
05920 Trans: downdateQUD:<s> => loadPlan:<s>
05930 Trans: loadPlan:<s> => execPlan:<s>
06010 Trans: execPlan:<s> => init:<s>
06020 Log: transition applied: init
06020 Trans: init:<s> => grounding:<s>
06120 Log: transition applied: getLatestMove
06130 Log: usr quit
06130 Trans: grounding:<s> => [says] => integrate:<s>
06180 Log: transition applied: integrateUsrQuit
06180 Trans: integrate:<s> => downdateQUD:<s>
06190 Trans: downdateQUD:<s> => loadPlan:<s>
06240 Trans: loadPlan:<s> => execPlan:<s>
06270 Trans: execPlan:<s> => select:<s>
06290 Trans: selectAction:<s> => selectMove:<s>
06330 Event: generateMove o(move:quit)
06330 Log: transition applied: selectOther
06330 Trans: selectMove:<s> => update:<s>
06380 Log: transition applied: init
06380 Trans: init:<s> => grounding:<s>
06400 Event: says o(move:quit speaker:sys)
06460 Log: transition applied: getLatestMove
06510 Log: sys quit
06510 Trans: grounding:<s> => [says] => integrate:<s>
06530 Event: top.Done o
06530 Event: top.Done o
06530 Log: transition applied: integrateSysQuit
06530 Trans: integrate:<s> => quit:<f>

```



## Appendix B

# Trace of MultiIBiS run: Agreement

```

00410 Event: driver.agree   o
00970 Trans: top:<cxml> => godis:<p>[listener addrAndObjMgmt update selectNoreply selectNormal
interpret generate questionSurveil generateStacker generator turnManager insideChart
trpChart outsideChart driver]
00990 Trans: driver:<s> => [driver.agree] => destinationAgree:<s>
01000 Event: driver.speak   o(contents:destination move:ask speaker:a)
01030 Event: startSpeaking   o(speaker:a)
01130 Log: added to DFs:   a
01140 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
01140 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
01210 Event: openTRP.free.comingUp   o(speaker:a)
01240 Event: utterance   o(contents:'What is the city of destination' speaker:a)
01260 Event: doSelect   o
01320 Event: says   o(contents:destination move:ask speaker:a)
01370 Trans: select.idle:<s> => [doSelect] => select.do:<s>
01370 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
01510 Event: driver.speak   o(contents:London move:answer speaker:b)
01550 Trans: select.selectAction:<s> => select.idle:<s>
01620 Event: printIS   o
01630 Log: =a=   ask:destination
01700 Event: integrated   o(contents:destination move:ask speaker:a)
01710 Log: TransTaken:   integrateOtherAskNoPlan
01710 Trans: integrate:<s> => [says] => downdateQUDefault:<s>
01800 Log: TransTaken:   downdateQUDefault
01800 Trans: downdateQUDefault:<s> => loadPlan:<s>
01850 Event: stopSpeaking   o(speaker:a)
01850 Log: TransTaken:   loadPlanDefault
01850 Trans: loadPlan:<s> => execPlan:<s>
01970 Log: TransTaken:   execPlanDefault
01970 Trans: execPlan:<s> => update:<s>
02020 Event: startSpeaking   o(speaker:b)
02090 Log: IS:   --BEGIN IS--
02150 Log: Registered DFs   [ a tSys ]
02160 Log: IS.private.agenda   []
02170 Log: IS.private.bel   [ destination # London ]
02180 Log: IS.private.plan.issue   none
02190 Log: IS.private.plan.openForMe   false
02200 Log: IS.private.plan.thePlan   []
02200 Log: IS.shared.com   []
02210 Log: max-qud   o(aset:[ ] issue:destination oset:[ ] spkr:a status:open )
02220 Log: IS.shared.qud   [ o( aet:[ ] issue:destination oset:[ ] spkr:a status:open ) ]
02230 Event: utterance   o(contents:'London.' speaker:b)
02230 Log: IS.shared.lu   o( moves:[ move( ask destination ) ] spkr:a )
02240 Log: IS.shared.pu   o( moves:[ ] spkr: unit )
02250 Log: IS:   ==END IS==
02370 Event: selectNoreply.do   o
02420 Log: close addressing status for issues owned by   a
02450 Trans: speaking:<s> => silence:<s>
02460 Trans: freeTRPComingUp:<s> => freeTRP:<s>
02470 Event: openTRP.free.here   o(speaker:a)
02540 Log: added to DFs:   b
02600 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
02600 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
02610 Event: openTRP.free.comingUp   o(speaker:b)
02680 Event: says   o(contents:London move:answer speaker:b)
02730 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>

```

```

02820 Event: doSelect   o
02840 Event: stopSpeaking   o(speaker:b)
02870 Event: doSelect   o
02990 Event: printIS   o
03000 Log: =b=      answer:london
03080 Event: integrated   o(contents:destination#london move:answer speaker:b)
03090 Log: TransTaken: integrateAnswerAgree
03090 Trans: integrate:<s> => [says] => downdateQUD:<s>
03180 Log: TransTaken: downdateQUDDefault
03180 Trans: downdateQUD:<s> => loadPlan:<s>
03230 Log: TransTaken: loadPlanDefault
03230 Trans: loadPlan:<s> => execPlan:<s>
03350 Log: TransTaken: execPlanDefault
03350 Trans: execPlan:<s> => update:<s>
03400 Trans: select.idle:<s> => [doSelect] => select.do:<s>
03400 Trans: select.Noreply.idle:<s> => [doSelect] => select.Noreply.active:<s>
03570 Trans: select.selectAction:<s> => select.selectMove:<s>
03580 Event: printIS   o
03640 Event: generate.generateMove   o(addresser:b contents:destination#london move:icm#(acc#pos))
03650 Log: TransTaken: selectPosAcceptanceICM
03840 Trans: select.selectMove:<s> => select.idle:<s>
03930 Log: close addressing status for issues owned by b
03960 Trans: speaking:<s> => silence:<s>
03970 Trans: freeTRPComingUp:<s> => freeTRP:<s>
03980 Event: openTRP.free.here   o(speaker:b)
03990 Trans: select.idle:<s> => [doSelect] => select.do:<s>
04130 Trans: select.selectAction:<s> => select.idle:<s>
04150 Log: IS: --BEGIN IS--
04160 Log: Registered DPs [ b a tSys ]
04170 Log: IS.private.agenda []
04180 Log: IS.private.bel [ destination # london ]
04190 Log: IS.private.plan.issue none
04200 Log: IS.private.plan.openForMe false
04210 Log: IS.private.plan.thePlan []
04210 Log: IS.shared.com [ destination # london ]
04220 Log: max-qud o( asset:[ a tSys ] issue:accept( destination # london ) issueObject:o(
asset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed ) oset:[ a tSys
] spkr:b status:closed )
04230 Log: IS.shared.qud [ o( asset:[ a tSys ] issue:accept( destination # london )
issueObject:o( asset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed
) oset:[ a tSys ] spkr:b status:closed ) o( asset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed ) ]
04240 Log: IS.shared.lu o( moves:[ move( answer london ) ] spkr:b )
04250 Log: IS.shared.pu o( moves:[ move( ask destination ) ] spkr:a )
04260 Log: IS: ==END IS==
04350 Log: IS: --BEGIN IS--
04360 Log: IS.private.agenda []
04360 Log: Registered DPs [ b a tSys ]
04370 Log: IS.private.bel [ destination # london ]
04380 Log: IS.private.plan.issue none
04390 Log: IS.private.plan.openForMe false
04400 Event: selectNoreply.do o
04400 Log: IS.private.plan.thePlan []
04410 Log: IS.shared.com [ destination # london ]
04420 Log: max-qud o( asset:[ a tSys ] issue:accept( destination # london ) issueObject:o(
asset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed ) oset:[ a tSys
] spkr:b status:closed )
04430 Log: IS.shared.lu o( moves:[ move( answer london ) ] spkr:b )
04430 Log: IS.shared.qud [ o( asset:[ a tSys ] issue:accept( destination # london )
issueObject:o( asset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed
) oset:[ a tSys ] spkr:b status:closed ) o( asset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed ) ]
04440 Log: IS.shared.pu o( moves:[ move( ask destination ) ] spkr:a )
04450 Log: IS: ==END IS==
04490 Trans: nothingToSay:<s> => wantToSpeak:<s>
04490 Trans: selectNoreply.active:<s> => [generate.generateMove] => selectNoreply.idle:<s>
04500 Trans: wantToSpeak:<s> => doGenerate:<s>
04560 Event: startSpeaking   o(speaker:tSys)
04570 Event: doSelect   o
04570 Event: generateNext   o
04620 Trans: iAmSilent:<s> => [startSpeaking] => iAmSpeaking:<s>
04620 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
04630 Event: openTRP.free.comingUp   o(speaker:tSys)
04690 Event: utterance o(contents:'Yes, London.' speaker:tSys)
04700 Event: says o(contents:destination#london move:icm#(acc#pos) speaker:tSys)
04720 Event: addressing o(addresser:b speaker:tSys)
04740 Trans: select.idle:<s> => [doSelect] => select.do:<s>
04740 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
04890 Trans: select.selectAction:<s> => select.idle:<s>
04900 Event: generateNext   o
04920 Event: doSelect   o
05100 Event: printIS   o
05110 Log: =tSys= icmacccpos:destinationlondon
05130 Event: integrated   o(contents:destination#london move:icm#(acc#pos) speaker:tSys)
05140 Log: TransTaken: integratePosAcceptICM
05140 Trans: integrate:<s> => [says] => downdateQUD:<s>
05180 Log: TransTaken: downdateQUD
05190 Log: removed: o( asset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed )
05320 Log: TransTaken: downdateQUDDefault
05320 Trans: downdateQUD:<s> => loadPlan:<s>

```

```

05360 Log: TransTaken:    loadPlanDefault
05360 Trans: loadPlan:<s> => execPlan:<s>
05420 Log: TransTaken:    execPlanDefault
05420 Trans: execPlan:<s> => update:<s>
05460 Log: update addr for issues owned by      tSys
05480 Event: closedTRP.comingup   o(speaker:tSys)
05480 Trans: freeTRPComingUp:<s> => [addressing] => othersTRPComingUp:<s>
05570 Event: stopSpeaking   o(speaker:tSys)
05580 Trans: doGenerate:<s> => [generateNext] => nothingToSay:<s>
05610 Trans: select.idle:<s> => [doSelect] => select.do:<s>
05710 Trans: select.selectAction:<s> => select.idle:<s>
05750 Log: IS:      --BEGIN IS--
05750 Log: Registered DPs      [ b a tSys ]
05760 Log: IS.private.agenda  [ ]
05820 Event: selectNoreply.do  o
05820 Log: IS.private.bel     [ destination # london ]
05830 Log: IS.private.plan.issue none
05840 Log: IS.private.plan.openForMe false
05850 Log: IS.private.plan.thePlan [ ]
05860 Log: IS.shared.com      [ destination # london ]
05870 Log: max-qud      o( asset:[ a tSys ] issue:accept( destination # london ) issueObject:o(
asset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed ) oset:[ a tSys
] spkr:b status:closed )
05880 Log: IS.shared.qud      [ o( asset:[ a tSys ] issue:accept( destination # london )
issueObject:o( asset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed
) oset:[ a tSys ] spkr:b status:closed ) ]
05890 Log: IS.shared.lu       o( moves:[ move( icm # accpos destination # london ) ] spkr:tSys
)
05890 Log: IS.shared.pu       o( moves:[ move( answer london ) ] spkr:b )
05900 Log: IS:      ==END IS==
05960 Log: Generate:   o(contents:destination#london move:icm#(acc#pos) speaker:tSys)#
removed.
06100 Log: close addressing status for issues owned by      tSys
06100 Trans: iAmSpeaking:<s> => [stopSpeaking] => iAmSilent:<s>
06110 Trans: othersTRPComingUp:<s> => othersTRP:<s>
06120 Event: closedTRP.here   o(speaker:tSys)
06140 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
09120 Event: othersTRPTimeOut  o
09130 Trans: othersTRP:<s> => [othersTRPTimeOut] => freeTRP:<s>
09140 Event: openTRP.free.here  o(speaker:tSys)
09150 Event: doSelect      o
09170 Trans: select.idle:<s> => [doSelect] => select.do:<s>
09170 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
09320 Trans: select.selectAction:<s> => select.idle:<s>

```



# Appendix C

## Trace of MultiIBiS run: Disagreement

```

00410 Event: driver.disagree   o
00970 Trans: top:<scxml> => godis:<p>[listener addrAndObjMgmt update selectNoreply selectNormal
interpret generate questionSurveil generateStacker generator turnManager insideChart
trpChart outsideChart driver]
00990 Trans: driver:<s> => [driver.disagree] => destinationDisagree:<s>
01000 Event: driver.speak   o(contents:destination move:ask speaker:a)
01030 Event: startSpeaking   o(speaker:a)
01130 Log: added to DPs:    a
01140 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
01140 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
01210 Event: openTRP.free.comingUp   o(speaker:a)
01240 Event: utterance   o(contents:'What is the city of destination' speaker:a)
01260 Event: doSelect   o
01320 Event: says   o(contents:destination move:ask speaker:a)
01440 Trans: select.idle:<s> => [doSelect] => select.do:<s>
01440 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
01510 Event: driver.speak   o(contents:london move:answer speaker:b)
01540 Trans: select.selectAction:<s> => select.idle:<s>
01670 Event: printIS   o
01680 Log: =a=      ask:destination
01700 Event: integrated   o(contents:destination move:ask speaker:a)
01710 Log: TransTaken: integrateOtherAskNoPlan
01710 Trans: integrate:<s> => [says] => downdateQUDDefault:<s>
01800 Log: TransTaken: downdateQUDDefault
01800 Trans: downdateQUDD:<s> => loadPlan:<s>
01840 Log: TransTaken: loadPlanDefault
01840 Trans: loadPlan:<s> => execPlan:<s>
01850 Event: stopSpeaking   o(speaker:a)
01960 Log: TransTaken: execPlanDefault
01960 Trans: execPlan:<s> => update:<s>
02020 Event: startSpeaking   o(speaker:b)
02140 Log: IS:      --BEGIN IS--
02150 Log: Registered DPs: [ a tSys ]
02160 Log: IS.private.agenda  []
02170 Log: IS.private.bel   [ destination # paris neg( destination # london ) ]
02170 Log: IS.private.plan.issue  none
02180 Log: IS.private.plan.openForMe  false
02190 Log: IS.private.plan.thePlan  []
02200 Log: IS.shared.com  []
02210 Log: max-qud   o( asset:[ ] issue:destination oset:[ ] spkr:a status:open )
02220 Event: utterance   o(contents:'London.' speaker:b)
02220 Log: IS.shared.qud  [ o( asset:[ ] issue:destination oset:[ ] spkr:a status:open ) ]
02230 Log: IS.shared.lu   o( moves:[ move( ask destination ) ] spkr:a )
02240 Log: IS.shared.pu   o( moves:[ ] spkr: unit )
02240 Log: IS:      ==END IS==
02420 Log: close addressing status for issues owned by     a
02440 Event: selectNoreply.do   o
02440 Trans: speaking:<s> => silence:<s>
02460 Event: openTRP.free.here   o(speaker:a)
02460 Trans: freeTRPComingUp:<s> => freeTRP:<s>
02580 Log: added to DPs:    b

```

```

02600 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
02600 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
02610 Event: openTRP.free.comingUp o(speaker:b)
02670 Event: says o(contents:london move:answer speaker:b)
02720 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
02820 Event: doSelect o
02840 Event: stopSpeaking o(speaker:b)
02870 Event: doSelect o
03060 Event: printIS o
03070 Log: =b= answer:london
03100 Event: integrated o(contents:london move:answer speaker:b)
03100 Log: TransTaken: integrateAnswerDisagree
03100 Trans: integrate:<s> => [says] => downdateQUD:<s>
03190 Log: TransTaken: downdateQUDDefault
03190 Trans: downdateQUD:<s> => loadPlan:<s>
03290 Log: TransTaken: loadPlanDefault
03290 Trans: loadPlan:<s> => execPlan:<s>
03360 Log: TransTaken: execPlanDefault
03360 Trans: execPlan:<s> => update:<s>
03410 Trans: select.idle:<s> => [doSelect] => select.do:<s>
03410 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
03570 Log: TransTaken: selectRespond
03570 Trans: select.selectAction:<s> => select.selectMove:<s>
03580 Event: printIS o
03650 Event: generate.generateMove o(addressee:b contents:destination#london move:icm#(acc#neg))
03660 Log: TransTaken: selectNegAnswerAcceptanceICM
03710 Event: generate.generateMove o(addressee:a contents:destination#paris move:answer)
03720 Log: TransTaken: selectAnswer
03910 Trans: select.selectMove:<s> => select.idle:<s>
04050 Log: close addressing status for issues owned by b
04080 Trans: speaking:<s> => silence:<s>
04090 Trans: freeTRPComingUp:<s> => freeTRP:<s>
04100 Event: openTRP.free.here o(speaker:b)
04120 Trans: select.idle:<s> => [doSelect] => select.do:<s>
04200 Log: TransTaken: selectRespond
04200 Trans: select.selectAction:<s> => select.selectMove:<s>
04210 Event: printIS o
04240 Event: generate.generateMove o(addressee:a contents:destination#paris move:answer)
04300 Log: TransTaken: selectAnswer
04420 Event: selectNoreply.do o
04420 Trans: select.selectMove:<s> => select.idle:<s>
04440 Log: IS: --BEGIN IS--
04450 Log: Registered DPs [ b a tSys ]
04460 Log: IS.private.agenda []
04470 Log: IS.private.bel [ destination # paris neg( destination # london ) ]
04480 Log: IS.private.plan.issue none
04490 Log: IS.private.plan.openForMe false
04550 Log: IS.private.plan.thePlan []
04560 Log: IS.shared.com [ destination # london ]
04570 Log: IS.shared.qud [ o( asset:[ a tSys ] issue:accept( destination # london )
issueObject:o( asset:[ a tSys ] issue:destination oset:[ ] spkr:a status:closed
) oset:[ a tSys ] spkr:b status:closed ) o( asset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed ) ]
max-qud o( asset:[ a tSys ] issue:accept( destination # london ) issueObject:o(
asset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed ) oset:[ a tSys ]
] spkr:b status:closed )
04580 Log: IS.shared.lu o( moves:[ move( answer london ) ] spkr:b )
04590 Log: IS.shared.pu o( moves:[ move( ask destination ) ] spkr:a )
04600 Log: IS: ==END IS==
04640 Log: IS: --BEGIN IS--
04650 Log: IS.private.agenda []
04650 Log: Registered DPs [ b a tSys ]
04660 Log: IS.private.bel [ destination # paris neg( destination # london ) ]
04670 Log: IS.private.plan.issue none
04680 Log: IS.private.plan.openForMe false
04690 Log: IS.private.plan.thePlan []
04700 Log: IS.shared.com [ destination # london ]
04710 Log: max-qud o( asset:[ a tSys ] issue:accept( destination # london ) issueObject:o(
asset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed ) oset:[ a tSys ]
] spkr:b status:closed )
04720 Log: IS.shared.lu o( moves:[ move( answer london ) ] spkr:b )
04720 Log: IS.shared.qud [ o( asset:[ a tSys ] issue:accept( destination # london )
issueObject:o( asset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed
) oset:[ a tSys ] spkr:b status:closed ) o( asset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed ) ]
04730 Log: IS.shared.pu o( moves:[ move( ask destination ) ] spkr:a )
04790 Log: IS: ==END IS==
04830 Trans: selectNoreply.active:<s> => [generate.generateMove] => selectNoreply.idle:<s>
04840 Trans: nothingToSay:<s> => wantToSpeak:<s>
04850 Event: startSpeaking o(speaker:tSys)
04850 Trans: wantToSpeak:<s> => doGenerate:<s>
04860 Event: generateNext o
04890 Event: doSelect o
04900 Log: IS: --BEGIN IS--
04910 Log: Registered DPs [ b a tSys ]
04920 Log: IS.private.agenda []
04930 Log: IS.private.bel [ destination # paris neg( destination # london ) ]
04940 Log: IS.private.plan.issue none
04940 Log: IS.private.plan.openForMe false
04950 Log: IS.private.plan.thePlan []

```

```

04960 Log: IS.shared.com      [ destination # london ]
04970 Log: max-qud      o( asset:[ a tSys ] issue:accept( destination # london ) issueObject:o(
04970 Log:   asset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed ) oset:[ a tSys
04970 Log:   ] spkr:b status:closed )
05040 Log: IS.shared.lu      o( moves:[ move( answer london ) ] spkr:b )
05040 Log: IS.shared.qud      [ o( asset:[ a tSys ] issue:accept( destination # london )
05040 Log:   issueObject:o( asset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed
05040 Log:   ) oset:[ a tSys ] spkr:b status:closed ) o( asset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed )
05050 Log: IS.shared.pu      o( moves:[ move( ask destination ) ] spkr:a )
05060 Log: IS:    ==END IS==
05130 Trans: iAmSilent:<s> => [startSpeaking] => iAmSpeaking:<s>
05130 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
05140 Event: openTRP.free.comingUp      o(speaker:tSys)
05200 Event: says      o(contents:destination#london move:icm#(acc#neg) speaker:tSys)
05200 Event: utterance      o(contents:'No, not London.' speaker:tSys)
05280 Event: addressing      o(addressee:b speaker:tSys)
05300 Trans: select.idle:<s> => [doSelect] => select.do:<s>
05300 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
05400 Log: TransTaken:  selectRespond
05400 Trans: select.selectAction:<s> => select.selectMove:<s>
05410 Event: generateNext      o
05410 Event: printIS      o
05440 Event: generate.generateMove      o(addressee:a contents:destination#paris move:answer)
05450 Log: TransTaken:  selectAnswer
05630 Trans: select.selectMove:<s> => select.idle:<s>
05660 Event: doSelect      o
05830 Event: printIS      o
05840 Log: =tSys= icmaccneg:destinationlondon
05870 Event: integrated      o(contents:destination#london move:icm#(acc#neg) speaker:tSys)
05880 Log: TransTaken:  integrateNegAcceptICM
05880 Trans: integrate:<s> => [says] => downdateQUDDefault:<s>
05950 Log: TransTaken:  downdateQUDDefault
05950 Trans: downdateQUD:<s> => loadPlan:<s>
06050 Log: TransTaken:  loadPlanDefault
06050 Trans: loadPlan:<s> => execPlan:<s>
06110 Log: TransTaken:  execPlanDefault
06110 Trans: execPlan:<s> => update:<s>
06150 Log: update addr for issues owned by      tSys
06160 Trans: freeTRPComingUp:<s> => [addressing] => othersTRPComingUp:<s>
06170 Event: closedTRP.comingup      o(speaker:tSys)
06210 Log: IS:    --BEGIN IS--
06210 Log: Registered DPs      [ b a tSys ]
06220 Log: IS.private.agenda      []
06230 Log: IS.private.bel      [ destination # paris neg( destination # london ) ]
06300 Log: IS.private.plan.issue      none
06310 Event: selectNoreply.do      o
06310 Log: IS.private.plan.openForMe      false
06310 Log: IS.private.plan.thePlan      []
06320 Log: IS.shared.com      []
06330 Log: max-qud      o( asset:[ a tSys ] issue:accept( destination # london ) issueObject:o(
06330 Log:   asset:[ tSys ] issue:destination oset:[ b ] spkr:a status:closed ) oset:[ a
06330 Log:   tSys ] spkr:b status:closed )
06340 Log: IS.shared.qud      [ o( asset:[ a tSys ] issue:accept( destination # london )
06340 Log:   issueObject:o( asset:[ tSys ] issue:destination oset:[ b ] spkr:a status:closed
06340 Log:   ) oset:[ a tSys ] spkr:b status:closed ) o( asset:[ tSys ] issue:destination oset:[ b ] spkr:a status:closed )
06350 Log: IS.shared.lu      o( moves:[ move( icm # accneg destination # london ) ] spkr:tSys
06350 Log: )
06360 Log: IS.shared.pu      o( moves:[ move( answer london ) ] spkr:b )
06370 Log: IS:    ==END IS==
06420 Event: utterance      o(contents:'I am going to Paris.' speaker:tSys)
06430 Event: says      o(contents:destination#paris move:answer speaker:tSys)
06450 Event: addressing      o(addressee:a speaker:tSys)
06550 Log: Generate:  o(addressee:a contents:destination#paris move:answer) '# rejected.'
06550 Trans: selectNoreply.active:<s> => [generate.generateMove] => selectNoreply.idle:<s>
06580 Trans: select.idle:<s> => [doSelect] => select.do:<s>
06580 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
06630 Event: generateNext      o
06680 Event: printIS      o
06680 Log: TransTaken:  selectRespond
06680 Trans: select.selectAction:<s> => select.selectMove:<s>
06720 Event: generate.generateMove      o(addressee:a contents:destination#paris move:answer)
06730 Log: TransTaken:  selectAnswer
06910 Trans: select.selectMove:<s> => select.idle:<s>
06940 Log: IS:    --BEGIN IS--
06950 Log: IS.private.agenda      []
06950 Log: Registered DPs      [ b a tSys ]
06960 Log: IS.private.bel      [ destination # paris neg( destination # london ) ]
06970 Log: IS.private.plan.issue      none
06980 Log: IS.private.plan.openForMe      false
06990 Log: IS.private.plan.thePlan      []
07000 Log: IS.shared.com      []
07060 Log: max-qud      o( asset:[ a tSys ] issue:accept( destination # london ) issueObject:o(
07060 Log:   asset:[ tSys ] issue:destination oset:[ b ] spkr:a status:closed ) oset:[ a
07060 Log:   tSys ] spkr:b status:closed )
07070 Log: IS.shared.qud      [ o( asset:[ a tSys ] issue:accept( destination # london )
07070 Log:   issueObject:o( asset:[ tSys ] issue:destination oset:[ b ] spkr:a status:closed
07070 Log:   ) oset:[ a tSys ] spkr:b status:closed ) o( asset:[ tSys ] issue:destination oset:[ b ] spkr:a status:closed )
07070 Log: IS.shared.lu      o( moves:[ move( icm # accneg destination # london ) ] spkr:tSys

```

```

)
07090 Log: IS.shared.pu o( moves:[ move( answer london ) ] spkr:b )
07090 Log: IS: ==END IS==
07140 Log: Generate: o(contents:destination#london move:icm#(acc#neg) speaker:tSys)#
removed.'
07180 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
07330 Event: printIS o
07340 Log: =tSys= answer:destinationparis
07370 Event: integrated o(contents:destination#paris move:answer speaker:tSys)
07370 Log: TransTaken: integrateAnswer
07370 Trans: integrate:<s> => [says] => downdateQUDDefault:<s>
07450 Log: TransTaken: downdateQUDDefault
07450 Trans: downdateQUDD:<s> => loadPlan:<s>
07490 Log: TransTaken: loadPlanDefault
07490 Trans: loadPlan:<s> => execPlan:<s>
07580 Event: selectNoreply.do o
07610 Log: TransTaken: execPlanDefault
07610 Trans: execPlan:<s> => update:<s>
07650 Log: update addr for issues owned by tSys
07700 Event: stopSpeaking o(speaker:tSys)
07700 Trans: doGenerate:<s> => [generateNext] => nothingToSay:<s>
07740 Log: IS: --BEGIN IS--
07750 Log: Registered DPs [ b a tSys ]
07810 Log: IS.private.agenda []
07820 Log: IS.private.bel [ destination # paris neg( destination # london ) ]
07830 Log: IS.private.plan.issue none
07840 Log: IS.private.plan.openForMe false
07850 Log: IS.private.plan.thePlan []
07850 Log: IS.shared.com [ destination # paris ]
07860 Log: max-qud o( aset:[ a tSys ] issue:accept( destination # paris ) issueObject:o(
aset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed ) oset:[ ] spkr:tSys
status:closed )
07870 Log: IS.shared.qud [ o( aset:[ a tSys ] issue:accept( destination # paris )
issueObject:o( aset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed
) oset:[ ] spkr:tSys status:closed ) o( aset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed ) o( aset:[ a tSys ] issue:accept( destination # london ) ] spkr:tSys )
07880 Log: IS.shared.lu o( moves:[ move( answer destination # paris ) move( icm # accneg
destination # london ) ] spkr:tSys )
07890 Log: IS.shared.pu o( moves:[ move( answer london ) ] spkr:b )
07900 Log: IS: ==END IS==
07950 Log: Generate: o(addressee:a contents:destination#paris move:answer)#
rejected.'
07990 Log: --BEGIN IS--
08050 Log: Registered DPs [ b a tSys ]
08060 Log: IS.private.agenda []
08070 Log: IS.private.bel [ destination # paris neg( destination # london ) ]
08080 Log: IS.private.plan.issue none
08090 Log: IS.private.plan.openForMe false
08100 Log: IS.private.plan.thePlan []
08110 Log: IS.shared.com [ destination # paris ]
08110 Log: max-qud o( aset:[ a tSys ] issue:accept( destination # paris ) issueObject:o(
aset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed ) oset:[ ] spkr:tSys
status:closed )
08120 Log: IS.shared.qud [ o( aset:[ a tSys ] issue:accept( destination # paris )
issueObject:o( aset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed
) oset:[ ] spkr:tSys status:closed ) o( aset:[ tSys ] issue:destination oset:[ ] spkr:a status:closed ) o( aset:[ a tSys ] issue:accept( destination # london ) ] spkr:tSys )
08130 Log: IS.shared.lu o( moves:[ move( answer destination # paris ) move( icm # accneg
destination # london ) ] spkr:tSys )
08140 Log: IS.shared.pu o( moves:[ move( answer london ) ] spkr:b )
08150 Log: IS: ==END IS==
08200 Log: Generate: o(contents:destination#paris move:answer speaker:tSys)#
removed.'
08350 Log: close addressing status for issues owned by tSys
08350 Trans: iAmSpeaking:<s> => [stopSpeaking] => iamSilent:<s>
08360 Trans: othersTRPComingUp:<s> => othersTRP:<s>
08370 Event: closedTRP.here o(speaker:tSys)

```

# Appendix D

## Trace of MultiIBiS run: Addressing

```

00400 Event: driver.addressing1      o
00970 Trans: top:<scxml> => godis:<p>[listener addrAndObjMgmt update selectNoreply selectNormal
interpret generate questionSurveil generateStacker generator turnManager insideChart
trpChart outsideChart driver]
00990 Trans: driver:<s> => [driver.addressing1] => addressing1:<s>
01000 Event: driver.speak      o(addrressee:b contents:travelPrice move:ask speaker:a)
01030 Event: startSpeaking    o(speaker:a)
01140 Log: added to DPs:      a
01210 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
01210 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
01220 Event: openTRP.free.comingUp   o(speaker:a)
01240 Event: utterance      o(contents:'Travel price information please.' speaker:a)
01270 Event: doSelect      o
01330 Event: says      o(contents:travelPrice move:ask speaker:a)
01350 Event: addressing    o(addrressee:b speaker:a)
01380 Trans: select.idle:<s> => [doSelect] => select.do:<s>
01380 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
01560 Trans: select.selectAction:<s> => select.idle:<s>
01620 Event: printIS      o
01630 Log: =a=      ask:travelPrice
01700 Event: integrated    o(contents:travelPrice move:ask speaker:a)
01710 Log: TransTaken:      integrateOtherAsk
01710 Trans: integrate:<s> => [says] => downdateQUDefault:<s>
01800 Log: TransTaken:      downdateQUDefault
01800 Trans: downdateQUDefault:<s> => loadPlan:<s>
01860 Event: stopSpeaking   o(speaker:a)
01870 Log: TransTaken:      findPlan
01870 Trans: loadPlan:<s> => execPlan:<s>
01990 Log: TransTaken:      execPlanDefault
01990 Trans: execPlan:<s> => update:<s>
02060 Log: update addr for issues owned by      a
02070 Log: added to DPs:      b
02090 Event: closedTRP.comingup   o(speaker:a)
02090 Trans: freeTRPComingUp:<s> => [addressing] => othersTRPComingUp:<s>
02200 Log: IS:      --BEGIN IS--
02210 Log: Registered DPs      [ b a tSys ]
02220 Log: IS.private.agenda    []
02230 Log: IS.private.bel      []
02230 Log: IS.private.plan.issue  travelPrice
02240 Log: IS.private.plan.openForMe  uncertain
02250 Log: IS.private.plan.thePlan  [ findout( howTravel a ) findout( departure a )
findout( destination a ) findout( travelMonth a ) findout( class a ) consultDB(
travelPrice ) ]
02260 Log: IS.shared.com      []
02270 Log: max-qud      o( asset:[ b ] issue:travelPrice oset:[ b ] spkr:a status:open )
02280 Log: IS.shared.qud      [ o( asset:[ b ] issue:travelPrice oset:[ b ] spkr:a status:open
) ]
02290 Log: IS.shared.lu      o( moves:[ move( ask travelPrice ) ] spkr:a )
02300 Log: IS.shared.pu      o( moves:[ ] spkr: unit )
02300 Log: IS:      ==END IS==
02380 Event: selectNoreply.do  o
02480 Log: close addressing status for issues owned by      a
02510 Trans: speaking:<s> => silence:<s>
02520 Trans: othersTRPComingUp:<s> => othersTRP:<s>
02530 Event: closedTRP.here    o(speaker:a)
02610 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>

```

```

05530 Event: othersTRPTimeOut      o
05540 Trans: othersTRP:<s> => [othersTRPTimeOut] => freeTRP:<s>
05550 Event: openTRP.free.here    o(speaker:a)
05560 Event: doSelect      o
05580 Trans: select.idle:<s> => [doSelect] => select.do:<s>
05580 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
05660 Log: plan: notForMe
05790 Trans: select.selectAction:<s> => select.idle:<s>
06590 Event: selectNoreply.do    o
06610 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
07010 Event: driver.speak      o(contents:travelPrice move:ask speaker:a)
07020 Event: startSpeaking     o(speaker:a)
07160 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
07160 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
07170 Event: openTRP.free.comingUp o(speaker:a)
07220 Event: doSelect      o
07230 Event: utterance      o(contents:'Travel price information please.' speaker:a)
07270 Trans: select.idle:<s> => [doSelect] => select.do:<s>
07270 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
07430 Trans: select.selectAction:<s> => select.idle:<s>
07490 Event: says      o(contents:travelPrice move:ask speaker:a)
07550 Event: printIS      o
07560 Log: =a= ask:travelPrice
07580 Event: integrated      o(contents:travelPrice move:ask speaker:a)
07640 Log: TransTaken: integrateOtherAsk
07640 Trans: integrate:<s> => [says] => downdateQUD:<s>
07730 Log: TransTaken: downdateQUDefault
07730 Trans: downdateQUD:<s> => loadPlan:<s>
07800 Log: TransTaken: findPlan
07800 Trans: loadPlan:<s> => execPlan:<s>
07870 Event: stopSpeaking     o(speaker:a)
07930 Log: TransTaken: execPlanDefault
07930 Trans: execPlan:<s> => update:<s>
07980 Log: IS: --BEGIN IS--
07990 Log: Registered DPs [ b a tSys ]
08000 Log: IS.private.agenda []
08010 Log: IS.private.bel []
08020 Log: IS.private.plan.issue travelPrice
08020 Log: IS.private.plan.openForMe uncertain
08030 Log: IS.private.plan.thePlan [ findout( howTravel a ) findout( departure a )
findout( destination a ) findout( travelMonth a ) findout( class a ) consultDB(
travelPrice ) ]
08040 Log: IS.shared.com []
08100 Log: max-qud o( asset:[ ] issue:travelPrice oset:[ ] spkr:a status:open )
08110 Log: IS.shared.qud [ o( asset:[ ] issue:travelPrice oset:[ ] spkr:a status:open
) o( asset:[ b ] issue:travelPrice oset:[ b ] spkr:a status:closed ) ]
08120 Log: IS.shared.lu o( moves:[ move( ask travelPrice ) ] spkr:a )
08130 Log: IS.shared.pu o( moves:[ move( ask travelPrice ) ] spkr:a )
08130 Log: IS: ==END IS==
08260 Log: close addressing status for issues owned by a
08270 Event: selectNoreply.do o
08340 Trans: speaking:<s> => silence:<s>
08350 Trans: freeTRPComingUp:<s> => freeTRP:<s>
08360 Event: openTRP.free.here    o(speaker:a)
08380 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
08400 Event: doSelect      o
08410 Trans: select.idle:<s> => [doSelect] => select.do:<s>
08410 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
08470 Log: plan: indeedForMe
08630 Event: printIS      o
08630 Log: Added action to agenda: findout(howTravel a)
08630 Trans: select.selectAction:<s> => select.selectMove:<s>
08800 Event: generate.generateMove o(addressee:a contents:howTravel move:ask)
08820 Log: TransTaken: selectAsk
08940 Trans: select.selectMove:<s> => select.idle:<s>
08960 Log: IS: --BEGIN IS--
08970 Log: Registered DPs [ b a tSys ]
08980 Log: IS.private.agenda []
08980 Log: IS.private.bel []
08990 Log: IS.private.plan.issue travelPrice
09060 Log: IS.private.plan.openForMe (true#'( '#'))
09070 Log: IS.private.plan.thePlan [ findout( howTravel a ) findout( departure a )
findout( destination a ) findout( travelMonth a ) findout( class a ) consultDB(
travelPrice ) ]
09070 Log: IS.shared.com []
09080 Log: max-qud o( asset:[ b tSys ] issue:travelPrice oset:[ ] spkr:a status:closed )
09090 Log: IS.shared.qud [ o( asset:[ b tSys ] issue:travelPrice oset:[ ] spkr:a status:closed
) o( asset:[ b ] issue:travelPrice oset:[ b ] spkr:a status:closed ) ]
09100 Log: IS.shared.lu o( moves:[ move( ask travelPrice ) ] spkr:a )
09110 Log: IS.shared.pu o( moves:[ move( ask travelPrice ) ] spkr:a )
09120 Log: IS: ==END IS==
09160 Trans: noQuestions:<s> => [generate.generateMove] => questions:<s>
09160 Trans: nothingToSay:<s> => wantToSpeak:<s>
09160 Trans: selectNoreply.active:<s> => [generate.generateMove] => selectNoreply.idle:<s>
09170 Trans: wantToSpeak:<s> => doGenerate:<s>
09180 Event: generateNext      o
09180 Event: startSpeaking     o(speaker:tSys)
09230 Trans: iAmSilent:<s> => [startSpeaking] => iAmSpeaking:<s>

```

```

09230 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
09240 Event: openTRP.free.comingUp   o(speaker:tSys)
09340 Event: utterance   o(contents:'How do you want to travel?' speaker:tSys)
09350 Event: says   o(contents:howTravel move:ask speaker:tSys)
09370 Event: addressing   o(addressee:a speaker:tSys)
09400 Event: doSelect   o
09420 Event: selectNoreply.do   o
09470 Event: printIS   o
09480 Log: =tSys= ask:howTravel
09550 Event: generateNext   o
09550 Event: integrated   o(contents:howTravel move:ask speaker:tSys)
09560 Log: TransTaken: integrateMeAsk
09560 Trans: integrate:<s> => [says] => downdateQUD:<s>
09560 Trans: questions:<s> => [says] => questionsAsked:<s>
09600 Log: TransTaken: downdateQUD
09610 Log: removed:   o( asset:[ b tSys ] issue:travelPrice oset:[ ] spkr:a status:closed )
09690 Log: TransTaken: downdateQUDDefault
09690 Trans: downdateQUD:<s> => loadPlan:<s>
09730 Log: TransTaken: loadPlanDefault
09730 Trans: loadPlan:<s> => execPlan:<s>
09840 Log: TransTaken: execPlanDefault
09840 Trans: execPlan:<s> => update:<s>
09890 Log: update addr for issues owned by   tSys
09900 Trans: freeTRPComingUp:<s> => [addressing] => othersTRPComingUp:<s>
09910 Event: closedTRP.comingup   o(speaker:tSys)
09940 Trans: select.idle:<s> => [doSelect] => select.do:<s>
09940 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
10070 Log: Item is already in qud   howTravel
10070 Trans: select.selectAction:<s> => select.idle:<s>
10100 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
10130 Log: IS: --BEGIN IS--
10140 Log: Registered DPs   [ b a tSys ]
10150 Log: IS.private.agenda   [ ]
10160 Log: IS.private.bel   [ ]
10170 Log: IS.private.plan.issue   travelPrice
10180 Log: IS.private.plan.openForMe   (true#'( ')#')
10190 Log: IS.private.plan.thePlan   [ findout( howTravel a ) findout( departure a )
findout( destination a ) findout( travelMonth a ) findout( class a ) consultDB(
travelPrice ) ]
10190 Log: IS.shared.com   [ ]
10200 Log: max-qud   o( asset:[ a ] issue:howTravel oset:[ a ] spkr:tSys status:open )
10210 Log: IS.shared.qud   [ o( asset:[ a ] issue:howTravel oset:[ a ] spkr:tSys status:open
) o( asset:[ b ] issue:travelPrice oset:[ b ] spkr:a status:closed ) ]
10220 Log: IS.shared.lu   o( moves:[ move( ask howTravel ) ] spkr:tSys )
10230 Log: IS.shared.pu   o( moves:[ move( ask travelPrice ) ] spkr:a )
10300 Log: IS: ==END IS==
10330 Event: stopSpeaking   o(speaker:tSys)
10340 Trans: doGenerate:<s> => [generateNext] => nothingToSay:<s>
10390 Log: Generate:   o(contents:howTravel move:ask speaker:tSys)#' removed.'
10490 Log: close addressing status for issues owned by   tSys
10490 Trans: iAmSpeaking:<s> => [stopSpeaking] => iAmSilent:<s>
10490 Trans: questionsAsked:<s> => [stopSpeaking] => noQuestions:<s>
10560 Event: closedTRP.here   o(speaker:tSys)
10560 Trans: othersTRPComingUp:<s> => othersTRP:<s>
10950 Event: selectNoreply.do   o
13560 Event: othersTRPTimeOut   o
13570 Trans: othersTRP:<s> => [othersTRPTimeOut] => freeTRP:<s>
13580 Event: openTRP.free.here   o(speaker:tSys)
13600 Event: doSelect   o
13610 Trans: select.idle:<s> => [doSelect] => select.do:<s>
13610 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
13690 Log: Item is already in qud   howTravel
13690 Trans: select.selectAction:<s> => select.idle:<s>
14630 Event: selectNoreply.do   o
14650 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>

```



## Appendix E

# Trace of MultiIBiS run: Discussion on the issue of Destination

```
00390 Event: driver.destination      o
00960 Trans: top:<scxml> => godis:<p>[listener addrAndObjMgmt update selectNoreply selectNormal
interpret generate questionSurveil generateStacker generator turnManager insideChart
trpChart outsideChart driver]
00980 Trans: driver:<s> => [driver.destination] => destinationDiscuss:<s>
00990 Event: driver.speak      o(addressee:b contents:destination move:ask speaker:a)
01040 Event: startSpeaking      o(speaker:a)
01200 Log: added to DPs:      a
01220 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
01220 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
01230 Event: openTRP.free.comingUp      o(speaker:a)
01250 Event: utterance      o(contents:'What is the city of destination' speaker:a)
01280 Event: doSelect      o
01340 Event: says      o(contents:destination move:ask speaker:a)
01430 Event: addressing      o(addressee:b speaker:a)
01460 Trans: select.idle:<s> => [doSelect] => select.do:<s>
01460 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
01570 Trans: select.selectAction:<s> => select.idle:<s>
01690 Event: printIS      o
01700 Log: =a=      ask:destination
01730 Event: integrated      o(contents:destination move:ask speaker:a)
01740 Log: TransTaken:      integrateOtherAskNoPlan
01740 Trans: integrate:<s> => [says] => downdateQUD:<s>
01820 Log: TransTaken:      downdateQUDDefault
01820 Trans: downdateQUD:<s> => loadPlan:<s>
01900 Event: stopSpeaking      o(speaker:a)
01920 Log: TransTaken:      loadPlanDefault
01920 Trans: loadPlan:<s> => execPlan:<s>
01990 Log: TransTaken:      execPlanDefault
01990 Trans: execPlan:<s> => update:<s>
02060 Log: update addr for issues owned by      a
02070 Log: added to DPs:      b
02130 Trans: freeTRPComingUp:<s> => [addressing] => othersTRPComingUp:<s>
02140 Event: closedTRP.comingup      o(speaker:a)
02200 Log: IS:      --BEGIN IS--
02210 Log: Registered DPs      [ b a tSys ]
02220 Log: IS.private.agenda      [ ]
02220 Log: IS.private.bel      [ ]
02230 Log: IS.private.plan.issue      none
02240 Log: IS.private.plan.openForMe      false
02250 Log: IS.private.plan.thePlan      [ ]
02260 Log: IS.shared.com      [ ]
02270 Log: max-qud      o( asset:[ b ] issue:destination oset:[ b ] spkr:a status:open )
02280 Log: IS.shared.qud      [ o( asset:[ b ] issue:destination oset:[ b ] spkr:a status:open
) ]
02290 Log: IS.shared.lu      o( moves:[ move( ask destination ) ] spkr:a )
02290 Log: IS.shared.pu      o( moves:[ ] spkr: unit )
02350 Log: IS:      ==END IS==
02460 Event: selectNoreply.do      o
02480 Log: close addressing status for issues owned by      a
02510 Trans: speaking:<s> => silence:<s>
02520 Trans: othersTRPComingUp:<s> => othersTRP:<s>
```

```

02580 Event: closedTRP.here o(speaker:a)
02610 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
03000 Event: driver.speak o(addressee:a contents:london move:answer speaker:b)
03010 Event: startSpeaking o(speaker:b)
03110 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
03110 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
03120 Event: openTRP.free.comingUp o(speaker:b)
03220 Event: doSelect o
03220 Event: utterance o(contents:'London.' speaker:b)
03280 Event: says o(contents:london move:answer speaker:b)
03330 Event: addressing o(addressee:a speaker:b)
03330 Trans: select.idle:<s> => [doSelect] => select.do:<s>
03330 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
03490 Trans: select.selectAction:<s> => select.idle:<s>
03690 Event: printIS o
03700 Log: =b= answer:london
03730 Event: integrated o(contents:london move:answer speaker:b)
03730 Log: TransTaken: integrateAnswer
03730 Trans: integrate:<s> => [says] => downdateQUD:<s>
03820 Log: TransTaken: downdateQUDDefault
03820 Trans: downdateQUD:<s> => loadPlan:<s>
03840 Event: stopSpeaking o(speaker:b)
03870 Log: TransTaken: loadPlanDefault
03870 Trans: loadPlan:<s> => execPlan:<s>
04000 Log: TransTaken: execPlanDefault
04000 Trans: execPlan:<s> => update:<s>
04060 Log: update addr for issues owned by b
04070 Trans: freeTRPComingUp:<s> => [addressing] => othersTRPComingUp:<s>
04080 Event: closedTRP.comingup o(speaker:b)
04190 Log: IS: --BEGIN IS--
04200 Log: Registered DPs [ b a tSys ]
04210 Log: IS.private.agenda []
04220 Log: IS.private.bel []
04230 Log: IS.private.plan.issue none
04240 Log: IS.private.plan.openForMe false
04250 Log: IS.private.plan.thePlan []
04260 Log: IS.shared.com [ destination # london ]
04260 Log: max-qud o( asset:[ a b ] issue:accept( destination # london ) issueObject:o(
asset:[ b ] issue:destination oset:[ ] spkr:a status:closed ) oset:[ ] spkr:b
status:closed )
04270 Log: IS.shared.qud [ o( asset:[ a b ] issue:accept( destination # london ) issueObject:o(
asset:[ b ] issue:destination oset:[ ] spkr:a status:closed ) oset:[ ] spkr:b
status:closed ) o( asset:[ b ] issue:destination oset:[ ] spkr:a status:closed ) ]
04280 Log: IS.shared.lu o( moves:[ move( answer london ) ] spkr:b )
04290 Log: IS.shared.pu o( moves:[ move( ask destination ) ] spkr:a )
04300 Log: IS: ==END IS==
04330 Event: selectNoreply.do o
04470 Log: close addressing status for issues owned by b
04500 Trans: speaking:<s> => silence:<s>
04520 Trans: othersTRPComingUp:<s> => othersTRP:<s>
04530 Event: closedTRP.here o(speaker:b)
04550 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
05010 Event: driver.speak o(addressee:b contents:unit move:icm#(acc#neg) speaker:a)
05030 Event: startSpeaking o(speaker:a)
05170 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
05170 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
05180 Event: openTRP.free.comingUp o(speaker:a)
05230 Event: doSelect o
05230 Event: utterance o(contents:'No.' speaker:a)
05290 Event: says o(contents:unit move:icm#(acc#neg) speaker:a)
05380 Event: addressing o(addressee:b speaker:a)
05390 Trans: select.idle:<s> => [doSelect] => select.do:<s>
05390 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
05500 Trans: select.selectAction:<s> => select.idle:<s>
05670 Event: printIS o
05680 Log: =a= (icm#(acc#neg)).unit
05700 Event: integrated o(contents:unit move:icm#(acc#neg) speaker:a)
05710 Log: TransTaken: integrateNegAcceptICM
05710 Trans: integrate:<s> => [says] => downdateQUD:<s>
05790 Log: TransTaken: downdateQUDAcceptance
05800 Log: removed: o(asset: issue:accept(destination#london) issueObject:o(asset:
issue:destination oset: spkr:a status:closed) oset: spkr:b status:closed)
05910 Event: stopSpeaking o(speaker:a)
05950 Log: TransTaken: downdateQUDDefault
05950 Trans: downdateQUD:<s> => loadPlan:<s>
06000 Log: TransTaken: loadPlanDefault
06000 Trans: loadPlan:<s> => execPlan:<s>
06070 Log: TransTaken: execPlanDefault
06070 Trans: execPlan:<s> => update:<s>
06190 Log: update addr for issues owned by a
06200 Trans: freeTRPComingUp:<s> => [addressing] => othersTRPComingUp:<s>
06210 Event: closedTRP.comingup o(speaker:a)
06260 Log: IS: --BEGIN IS--
06270 Log: Registered DPs [ b a tSys ]
06280 Log: IS.private.agenda []
06290 Log: IS.private.bel []
06300 Log: IS.private.plan.issue none
06310 Log: IS.private.plan.openForMe false

```

```

06320 Log: IS.private.plan.thePlan      [ ]
06320 Log: IS.shared.com      [ ]
06330 Log: max-qud      o( asset:[ b ] issue:destination oset:[ b ] spkr:a status:closed )
06340 Log: IS.shared.qud      [ o( asset:[ b ] issue:destination oset:[ b ] spkr:a status:closed )
) ]
06400 Event: selectNoreply.do      o
06400 Log: IS.shared.lu      o(moves:[ move(icm#(acc#neg)#unit)] spkr:a)
06410 Log: IS.shared.pu      o( moves:[ move( answer london ) ] spkr:b )
06420 Log: IS:    ==END IS==
06540 Log: close addressing status for issues owned by      a
06570 Trans: speaking:<s> => silence:<s>
06590 Event: closedTRP.here      o(speaker:a)
06590 Trans: othersTRPComingUp:<s> => othersTRP:<s>
06670 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
07020 Event: driver.speak      o(addressee:b contents:paris move:answer speaker:a)
07040 Event: startSpeaking      o(speaker:a)
07130 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
07130 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
07140 Event: openTRP.free.comingUp      o(speaker:a)
07240 Event: doSelect      o
07240 Event: utterance      o(contents:'Paris.' speaker:a)
07300 Event: says      o(contents:paris move:answer speaker:a)
07350 Trans: select.idle:<s> => [doSelect] => select.do:<s>
07350 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
07360 Event: addressing      o(addressee:b speaker:a)
07520 Trans: select.selectAction:<s> => select.idle:<s>
07660 Event: printIS      o
07670 Log: =a=      answer:paris
07750 Event: integrated      o(contents:paris move:answer speaker:a)
07760 Log: TransTaken:      integrateAnswer
07760 Trans: integrate:<s> => [says] => downdateQUD:<s>
07850 Log: TransTaken:      downdateQUDDefault
07850 Trans: downdateQUD:<s> => loadPlan:<s>
07860 Event: stopSpeaking      o(speaker:a)
07900 Log: TransTaken:      loadPlanDefault
07900 Trans: loadPlan:<s> => execPlan:<s>
08020 Log: TransTaken:      execPlanDefault
08020 Trans: execPlan:<s> => update:<s>
08090 Log: update addr for issues owned by      a
08100 Trans: freeTRPComingUp:<s> => [addressing] => othersTRPComingUp:<s>
08110 Event: closedTRP.comingup      o(speaker:a)
08170 Log: IS:    --BEGIN IS--
08180 Log: IS.private.agenda      [ ]
08180 Log: Registered DPs      [ b a tSys ]
08190 Log: IS.private.bel      [ ]
08250 Log: IS.private.plan.issue      none
08260 Log: IS.private.plan.openForMe      false
08270 Log: IS.private.plan.thePlan      [ ]
08280 Log: IS.shared.com      [ destination # paris ]
08290 Log: max-qud      o( asset:[ a b ] issue:accept( destination # paris ) issueObject:o(
asset:[ b ] issue:destination oset:[ ] spkr:a status:closed ) oset:[ ] spkr:a
status:closed )
08300 Log: IS.shared.qud      [ o( asset:[ b ] issue:accept( destination # paris ) issueObject:o(
asset:[ b ] issue:destination oset:[ ] spkr:a status:closed ) oset:[ ] spkr:a
status:closed ) o( asset:[ b ] issue:destination oset:[ ] spkr:a status:closed ) ]
08310 Log: IS.shared.lu      o( moves:[ move( answer paris ) ] spkr:a )
08320 Log: IS.shared.pu      o(moves:[ move(icm#(acc#neg)#unit)] spkr:a)
08350 Log: IS:    ==END IS==
08350 Event: selectNoreply.do      o
08500 Log: close addressing status for issues owned by      a
08530 Trans: speaking:<s> => silence:<s>
08540 Trans: othersTRPComingUp:<s> => othersTRP:<s>
08550 Event: closedTRP.here      o(speaker:a)
08580 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
09030 Event: driver.speak      o(addressee:a contents:unit move:icm#(acc#pos) speaker:b)
09040 Event: startSpeaking      o(speaker:b)
09190 Event: openTRP.free.comingUp      o(speaker:b)
09190 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
09190 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
09250 Event: doSelect      o
09250 Event: utterance      o(contents:'Yes.' speaker:b)
09310 Event: says      o(contents:unit move:icm#(acc#pos) speaker:b)
09360 Trans: select.idle:<s> => [doSelect] => select.do:<s>
09360 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
09520 Trans: select.selectAction:<s> => select.idle:<s>
09630 Event: printIS      o
09690 Log: =b=      (icm#(acc#pos)):unit
09710 Event: integrated      o(contents:unit move:icm#(acc#pos) speaker:b)
09720 Log: TransTaken:      integratePosAcceptICM
09720 Trans: integrate:<s> => [says] => downdateQUD:<s>
09770 Log: TransTaken:      downdateQUD
09780 Log: removed:      o( asset:[ b ] issue:destination oset:[ ] spkr:a status:closed )
09870 Event: stopSpeaking      o(speaker:b)
09870 Log: TransTaken:      downdateQUDAcceptance
09870 Log: removed:      o(asset: issue:accept(destination#paris) issueObject:o(asset:
issue:destination oset: spkr:a status:closed) oset: spkr:a status:closed)
10020 Log: TransTaken:      downdateQUDDefault
10020 Trans: downdateQUD:<s> => loadPlan:<s>

```

```

10070 Log: TransTaken:    loadPlanDefault
10070 Trans: loadPlan:<s> => execPlan:<s>
10140 Log: TransTaken:    execPlanDefault
10140 Trans: execPlan:<s> => update:<s>
10250 Log: IS:      --BEGIN IS--
10250 Log: Registered DPs [ b a tSys ]
10260 Log: IS.private.agenda [ ]
10270 Log: IS.private.bel [ ]
10280 Log: IS.private.plan.issue none
10290 Log: IS.private.plan.openForMe false
10300 Log: IS.private.plan.thePlan [ ]
10310 Log: IS.shared.com [ destination # paris ]
10310 Log: max-qud undefined
10320 Log: IS.shared.qud [ ]
10330 Log: IS.shared.lu o(moves:[ move(icm#(acc#pos)#unit)] spkr:b)
10340 Log: IS.shared.pu o( moves:[ move( answer paris ) ] spkr:a )
10350 Log: IS:      ==END IS==
10360 Event: selectNoreply.do o
10530 Log: close addressing status for issues owned by b
10560 Trans: speaking:<s> => silence:<s>
10570 Trans: freeTRPComingUp:<s> => freeTRP:<s>
10580 Event: openTRP.free.here o(speaker:b)
10600 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
10620 Event: doSelect o
10640 Trans: select.idle:<s> => [doSelect] => select.do:<s>
10640 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
10780 Trans: select.selectAction:<s> => select.idle:<s>
11650 Event: selectNoreply.do o
11670 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>

```

## Appendix F

# Trace of MultiIBiS run: Hotel and Travel price

A simulated user (`usr`) interacting with a simulated hotel-price agent (`hSys`) and a MultiIBiS travel-price agent (`tSys`). The user asks for hotel price information, then for travel information.

```

00410 Event: driver.price      o
00970 Trans: top:<scxml> => godis:<p>[listener addrAndObjMgmt update selectNoreply selectNormal
    interpret generate questionSurveil generateStacker generator turnManager insideChart
    trpChart outsideChart driver]
00990 Trans: driver:<s> => [driver.price] => hotel_travel:<s>
01000 Event: driver.speak      o(contents:hotelPrice move:ask speaker:usr)
01050 Event: startSpeaking     o(speaker:usr)
01210 Log: added to DPs:    usr
01220 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
01220 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
01230 Event: openTRP.free.comingUp   o(speaker:usr)
01260 Event: utterance        o(contents:'Hotel price information please.' speaker:usr)
01280 Event: doSelect         o
01340 Event: says            o(contents:hotelPrice move:ask speaker:usr)
01460 Trans: select.idle:<s> => [doSelect] => select.do:<s>
01460 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
01570 Trans: select.selectAction:<s> => select.idle:<s>
01790 Event: printIS          o
01790 Log: =usr= ask:hotelPrice
01820 Event: integrated       o(contents:hotelPrice move:ask speaker:usr)
01820 Log: TransTaken: defaultIntegrate
01830 Log: eventdata:        o(contents:hotelPrice move:ask speaker:usr)
01830 Trans: integrate:<s> => [says] => downdateQUD:<s>
01970 Log: TransTaken: downdateQUDDefault
01970 Trans: downdateQUD:<s> => loadPlan:<s>
02000 Event: driver.speak      o(contents:destination move:ask speaker:hSys)
02020 Log: TransTaken: loadPlanDefault
02020 Trans: loadPlan:<s> => execPlan:<s>
02090 Log: TransTaken: execPlanDefault
02090 Trans: execPlan:<s> => update:<s>
02190 Log: IS: --BEGIN IS--
02200 Log: Registered DPs:   [ usr tSys ]
02210 Log: IS.private.agenda  []
02220 Log: IS.private.bel    []
02220 Log: IS.private.plan.issue none
02230 Log: IS.private.plan.openForMe false
02240 Log: IS.private.plan.thePlan []
02250 Log: IS.shared.com     []
02260 Log: max-qud undefined
02270 Log: IS.shared.qud     []
02280 Log: IS.shared.lu      o( moves:[ notMeaningful ] spkr:usr )
02280 Log: IS.shared.pu      o( moves:[ ] spkr: unit )
02290 Log: IS: ==END IS==
02430 Event: startSpeaking    o(speaker:hSys)
02460 Event: selectNoreply.do o
02550 Log: added to DPs:    hSys

```

```

02560 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
02560 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
02620 Event: openTRP.free.comingUp o(speaker:hSys)
02640 Event: utterance o(contents:'What is the city of destination' speaker:hSys)
02680 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
02730 Event: doSelect o
02790 Event: says o(contents:destination move:ask speaker:hSys)
02890 Trans: select.idle:<s> => [doSelect] => select.do:<s>
02890 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
03000 Trans: select.selectAction:<s> => select.idle:<s>
03010 Event: driver.speak o(contents:london move:answer speaker:usr)
03110 Event: stopSpeaking o(speaker:usr)
03150 Event: printIS o
03160 Log: =hSys= ask:destination
03190 Event: integrated o(contents:destination move:ask speaker:hSys)
03200 Log: TransTaken: integrateOtherAskNoPlan
03200 Trans: integrate:<s> => [says] => downdateQUDDefault
03280 Log: TransTaken: downdateQUDDefault
03280 Trans: downdateQUDD:<s> => loadPlan:<s>
03380 Log: TransTaken: loadPlanDefault
03380 Trans: loadPlan:<s> => execPlan:<s>
03460 Log: TransTaken: execPlanDefault
03460 Trans: execPlan:<s> => update:<s>
03510 Event: startSpeaking o(speaker:usr)
03670 Log: close addressing status for issues owned by usr
03720 Event: utterance o(contents:'London.' speaker:usr)
03730 Log: IS: --BEGIN IS--
03740 Log: Registered DPs [ hSys usr tSys ]
03750 Log: IS.private.agenda []
03760 Log: IS.private.bel []
03770 Log: IS.private.plan.issue none
03830 Log: IS.private.plan.openForMe false
03840 Log: IS.private.plan.thePlan []
03850 Log: IS.shared.com []
03860 Log: max-qud o( asset:[ ] issue:destination oset:[ ] spkr:hSys status:open )
03870 Log: IS.shared.qud [ o( asset:[ ] issue:destination oset:[ ] spkr:hSys status:open ) ]
03880 Log: IS.shared.lu o( moves:[ move( ask destination ) ] spkr:hSys )
03890 Event: selectNoreply.do o
03890 Log: IS.shared.pu o( moves:[ notMeaningful ] spkr:usr )
03900 Log: IS: ==END IS==
04020 Event: driver.speak o(contents:hotelPrice#1234 move:answer speaker:hSys)
04020 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
04020 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
04080 Event: openTRP.free.comingUp o(speaker:usr)
04140 Event: says o(contents:london move:answer speaker:usr)
04200 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
04250 Event: startSpeaking o(speaker:hSys)
04370 Event: doSelect o
04450 Event: stopSpeaking o(speaker:hSys)
04460 Event: utterance o(contents:'The hotel price is 1234 Euro' speaker:hSys)
04600 Event: printIS o
04610 Log: =usr= answer:london
04640 Event: integrated o(contents:london move:answer speaker:usr)
04650 Log: TransTaken: integrateAnswer
04650 Trans: integrate:<s> => [says] => downdateQUDD:<s>
04740 Log: TransTaken: downdateQUDDDefault
04740 Trans: downdateQUDD:<s> => loadPlan:<s>
04840 Log: TransTaken: loadPlanDefault
04840 Trans: loadPlan:<s> => execPlan:<s>
04910 Log: TransTaken: execPlanDefault
04910 Trans: execPlan:<s> => update:<s>
05010 Event: openTRP.free.comingUp o(speaker:hSys)
05010 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
05010 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
05030 Event: driver.speak o(contents:travelPrice move:ask speaker:usr)
05120 Trans: select.idle:<s> => [doSelect] => select.do:<s>
05120 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
05230 Trans: select.selectAction:<s> => select.idle:<s>
05380 Log: close addressing status for issues owned by hSys
05440 Event: says o(contents:hotelPrice#1234 move:answer speaker:hSys)
05500 Log: IS: --BEGIN IS--
05500 Log: Registered DPs [ hSys usr tSys ]
05510 Log: IS.private.agenda []
05520 Log: IS.private.bel []
05530 Event: stopSpeaking o(speaker:usr)
05530 Log: IS.private.plan.issue none
05540 Log: IS.private.plan.openForMe false
05600 Log: IS.private.plan.thePlan []
05610 Log: IS.shared.com [ destination # london ]
05620 Log: max-qud o( asset:[ hSys ] issue:accept( destination # london ) issueObject:o(
asset:[ usr tSys ] issue:destination oset:[ ] spkr:hSys status:closed ) oset:[
usr tSys ] spkr:usr status:open )
05630 Log: IS.shared.qud [ o( asset:[ hSys ] issue:accept( destination # london ) issueObject:o(
asset:[ usr tSys ] issue:destination oset:[ ] spkr:hSys status:closed ) oset:[
usr tSys ] spkr:usr status:open ) o( asset:[ usr tSys ] issue:destination oset:[ ] spkr:hSys status:closed ) ]
05640 Log: IS.shared.lu o( moves:[ ] spkr:usr )
05640 Log: IS.shared.pu o( moves:[ move( answer london ) ] spkr:usr )

```

```

05650 Log: IS: ==END IS==
05730 Event: doSelect o
05840 Event: startSpeaking o(speaker:usr)
06100 Event: utterance o(contents:'Travel price information please.' speaker:usr)
06130 Event: printIS o
06130 Event: selectNoreply.do o
06140 Log: =hSys= answer:hotelPrice1234
06160 Event: integrated o(contents:hotelPrice#1234 move:answer speaker:hSys)
06170 Log: TransTaken: defaultIntegrate
06170 Log: eventdata: o(contents:hotelPrice#1234 move:answer speaker:hSys)
06170 Trans: integrate:<s> => [says] => downdateQUD:<s>
06230 Log: TransTaken: downdateQUD
06240 Log: removed: o( asset:[ usr tSys ] issue:destination oset:[ ] spkr:hSys status:closed
)
06370 Event: stopSpeaking o(speaker:hSys)
06380 Log: TransTaken: downdateQUDAcceptance
06390 Log: removed: o(asset: issue:accept(destination#london) issueObject:o(asset:
issue:destination oset: spkr:hSys status:closed) oset: spkr:usr status:open)
06480 Log: TransTaken: downdateQUDDefault
06480 Trans: downdateQUD:<s> => loadPlan:<s>
06530 Log: TransTaken: loadPlanDefault
06530 Trans: loadPlan:<s> => execPlan:<s>
06650 Log: TransTaken: execPlanDefault
06650 Trans: execPlan:<s> => update:<s>
06750 Log: close addressing status for issues owned by usr
06770 Trans: speaking:<s> => silence:<s>
06790 Event: openTRP.free.here o(speaker:hSys)
06790 Trans: freeTRPComingUp:<s> => freeTRP:<s>
06810 Trans: select.idle:<s> => [doSelect] => select.do:<s>
06950 Trans: select.selectAction:<s> => select.idle:<s>
07010 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
07010 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
07020 Event: openTRP.free.comingUp o(speaker:usr)
07130 Event: says o(contents:travelPrice move:ask speaker:usr)
07190 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
07240 Log: IS: --BEGIN IS--
07250 Log: IS.private.agenda []
07250 Log: Registered DPs [ hSys usr tSys ]
07260 Log: IS.private.bel []
07270 Log: IS.private.plan.issue none
07280 Log: IS.private.plan.openForMe false
07290 Log: IS.private.plan.thePlan []
07300 Log: IS.shared.com [ destination # london ]
07310 Log: max-qud undefined
07370 Log: IS.shared.qud []
07380 Log: IS.shared.lu o( moves:[ ] spkr:hSys )
07380 Log: IS.shared.pu o( moves:[ notMeaningful ] spkr:hSys )
07390 Log: IS: ==END IS==
07520 Log: close addressing status for issues owned by hSys
07630 Event: doSelect o
07680 Event: doSelect o
07770 Event: printIS o
07780 Log: =usr= ask:travelPrice
07800 Event: integrated o(contents:travelPrice move:ask speaker:usr)
07810 Log: TransTaken: integrateOtherAsk
07810 Trans: integrate:<s> => [says] => downdateQUD:<s>
07870 Event: stopSpeaking o(speaker:usr)
07950 Log: TransTaken: downdateQUDDefault
07950 Trans: downdateQUD:<s> => loadPlan:<s>
08020 Log: TransTaken: findPlan
08020 Trans: loadPlan:<s> => execPlan:<s>
08150 Log: TransTaken: execPlanDefault
08150 Trans: execPlan:<s> => update:<s>
08200 Trans: select.idle:<s> => [doSelect] => select.do:<s>
08200 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
08250 Log: plan: not certain
08250 Trans: select.selectAction:<s> => select.idle:<s>
08290 Trans: select.idle:<s> => [doSelect] => select.do:<s>
08390 Log: plan: not certain
08390 Trans: select.selectAction:<s> => select.idle:<s>
08440 Event: doSelect o
08440 Log: IS: --BEGIN IS--
08450 Log: Registered DPs [ hSys usr tSys ]
08460 Log: IS.private.agenda []
08470 Log: IS.private.bel []
08480 Log: IS.private.plan.issue travelPrice
08480 Log: IS.private.plan.openForMe uncertain
08490 Log: IS.private.plan.thePlan [ findout( howTravel usr ) findout( departure usr
) findout( destination usr ) findout( travelMonth usr ) findout( class usr
) consultDB( travelPrice ) ]
08500 Log: IS.shared.com [ destination # london ]
08510 Log: max-qud o( asset:[ ] issue:travelPrice oset:[ ] spkr:usr status:open )
08520 Log: IS.shared.qud [ o( asset:[ ] issue:travelPrice oset:[ ] spkr:usr status:open
) ]
08530 Log: IS.shared.lu o( moves:[ move( ask travelPrice ) ] spkr:usr )
08540 Log: IS.shared.pu o( moves:[ notMeaningful ] spkr:hSys )
08550 Log: IS: ==END IS==
08620 Event: doSelect o

```

```

08720 Log: close addressing status for issues owned by      usr
08750 Trans: speaking:<s> => silence:<s>
08760 Trans: freeTRPComingUp:<s> => freeTRP:<s>
08770 Event: openTRP.free.here   o(speaker:usr)
08780 Trans: select.idle:<s> => [doSelect] => select.do:<s>
08890 Log: plan: indeedForMe
09000 Event: printIS   o
09000 Log: Added action to agenda:   findout(howTravel usr)
09000 Trans: select.selectAction:<s> => select.selectMove:<s>
09180 Event: generate.generateMove   o(addresssee:usr contents:howTravel move:ask)
09190 Log: TransTaken: selectAsk
09200 Event: selectNoreply.do   o
09310 Trans: select.selectMove:<s> => select.idle:<s>
09320 Trans: select.idle:<s> => [doSelect] => select.do:<s>
09490 Event: printIS   o
09490 Log: Added action to agenda:   findout(howTravel usr)
09490 Trans: select.selectAction:<s> => select.selectMove:<s>
09670 Event: generate.generateMove   o(addresssee:usr contents:howTravel move:ask)
09680 Log: TransTaken: selectTask
09800 Trans: select.selectMove:<s> => select.idle:<s>
09820 Event: doSelect   o
09840 Log: IS: --BEGIN IS--
09850 Log: Registered DPs [ hSys usr tSys ]
09910 Log: IS.private.agenda [ ]
09920 Log: IS.private.bel [ ]
09930 Log: IS.private.plan.issue travelPrice
09930 Log: IS.private.plan.openForMe (true#'( '#'))
09940 Log: IS.private.plan.thePlan [ findout( howTravel usr ) findout( departure usr
) findout( destination usr ) findout( travelMonth usr ) findout( class usr
) consultDB( travelPrice ) ]
09950 Log: IS.shared.com [ destination # london ]
09960 Log: max-qud o( asset:[ hSys tSys ] issue:travelPrice oset:[ ] spkr:usr status:closed
)
09970 Log: IS.shared.qud [ o( asset:[ hSys tSys ] issue:travelPrice oset:[ ] spkr:usr
status:closed ) ]
09980 Log: IS.shared.lu o( moves:[ move( ask travelPrice ) ] spkr:usr )
09990 Log: IS.shared.pu o( moves:[ notMeaningful ] spkr:hSys )
10000 Log: IS: ==END IS==
10030 Trans: noQuestions:<s> => [generate.generateMove] => questions:<s>
10030 Trans: selectNoreply.active:<s> => [generate.generateMove] => selectNoreply.idle:<s>
10040 Trans: nothingToSay:<s> => wantToSpeak:<s>
10050 Trans: wantToSpeak:<s> => doGenerate:<s>
10060 Event: generateNext   o
10060 Event: startSpeaking   o(speaker:tSys)
10070 Log: IS: --BEGIN IS--
10080 Log: Registered DPs [ hSys usr tSys ]
10090 Log: IS.private.agenda [ ]
10150 Log: IS.private.bel [ ]
10160 Log: IS.private.plan.issue travelPrice
10170 Log: IS.private.plan.openForMe (true#'( '#'))
10180 Log: IS.private.plan.thePlan [ findout( howTravel usr ) findout( departure usr
) findout( destination usr ) findout( travelMonth usr ) findout( class usr
) consultDB( travelPrice ) ]
10190 Log: IS.shared.com [ destination # london ]
10200 Log: max-qud o( asset:[ hSys tSys ] issue:travelPrice oset:[ ] spkr:usr status:closed
)
10210 Log: IS.shared.qud [ o( asset:[ hSys tSys ] issue:travelPrice oset:[ ] spkr:usr
status:closed ) ]
10220 Log: IS.shared.lu o( moves:[ move( ask travelPrice ) ] spkr:usr )
10220 Log: IS.shared.pu o( moves:[ notMeaningful ] spkr:hSys )
10230 Log: IS: ==END IS==
10260 Trans: select.idle:<s> => [doSelect] => select.do:<s>
10260 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
10430 Event: printIS   o
10430 Log: Added action to agenda:   findout(howTravel usr)
10430 Trans: select.selectAction:<s> => select.selectMove:<s>
10540 Trans: select.selectMove:<s> => select.idle:<s>
10590 Event: openTRP.free.comingUp   o(speaker:tSys)
10590 Trans: iAmSilent:<s> => [startSpeaking] => iAmSpeaking:<s>
10590 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
10700 Event: utterance   o(contents:'How do you want to travel?' speaker:tSys)
10710 Event: says   o(contents:howTravel move:ask speaker:tSys)
10730 Event: addressing   o(addresssee:usr speaker:tSys)
10760 Log: IS: --BEGIN IS--
10770 Log: Registered DPs [ hSys usr tSys ]
10780 Log: IS.private.agenda [ findout( howTravel usr ) ]
10790 Log: IS.private.bel [ ]
10800 Log: IS.private.plan.issue travelPrice
10810 Log: IS.private.plan.openForMe (true#'( '#'))
10820 Log: IS.private.plan.thePlan [ findout( howTravel usr ) findout( departure usr
) findout( destination usr ) findout( travelMonth usr ) findout( class usr
) consultDB( travelPrice ) ]
10820 Log: IS.shared.com [ destination # london ]
10830 Log: max-qud o( asset:[ hSys tSys ] issue:travelPrice oset:[ ] spkr:usr status:closed
)
10840 Log: IS.shared.qud [ o( asset:[ hSys tSys ] issue:travelPrice oset:[ ] spkr:usr
status:closed ) ]
10850 Log: IS.shared.lu o( moves:[ ] spkr:usr )

```

```

10910 Log: IS.shared.pu      o( moves:[ move( ask travelPrice ) ] spkr:usr )
10920 Event: generateNext   o
10920 Log: IS:    ==END IS==
10950 Event: doSelect      o
11030 Event: printIS      o
11040 Log: =tSys=      ask:howTravel
11060 Event: integrated    o(contents:howTravel move:ask speaker:tSys)
11070 Event: driver.speak   o(contents:train move:answer speaker:usr)
11070 Log: TransTaken: integrateMeAsk
11070 Trans: integrate:<s> => [says] => downdateQUD:<s>
11070 Trans: questions:<s> => [says] => questionsAsked:<s>
11170 Log: TransTaken: downdateQUD
11180 Log: removed:   o( asset:[ hSys tSys ] issue:travelPrice oset:[ ] spkr:usr status:closed
11180 Log: )
11260 Event: selectNoreply.do   o
11260 Log: TransTaken: downdateQUDDefault
11260 Trans: downdateQUD:<s> => loadPlan:<s>
11300 Log: TransTaken: loadPlanDefault
11300 Trans: loadPlan:<s> => execPlan:<s>
11360 Log: TransTaken: execPlanDefault
11360 Trans: execPlan:<s> => update:<s>
11460 Log: update addr for issues owned by   tSys
11480 Event: closedTRP.comingup   o(speaker:tSys)
11480 Trans: freeTRPComingUp:<s> => [addressing] => othersTRPComingUp:<s>
11520 Event: stopSpeaking   o(speaker:tSys)
11520 Trans: doGenerate:<s> => [generateNext] => nothingToSay:<s>
11560 Trans: select.idle:<s> => [doSelect] => select.do:<s>
11730 Event: printIS      o
11730 Log: Added action to agenda:   findout(howTravel usr)
11730 Trans: select.selectAction:<s> => select.selectMove:<s>
11860 Trans: select.selectMove:<s> => select.idle:<s>
11900 Log: IS:    --BEGIN IS--
11900 Log: Registered DPs  [ hSys usr tSys ]
11910 Log: IS.private.agenda  [ findout( howTravel usr ) ]
11920 Log: IS.private.bel    [ ]
11930 Log: IS.private.plan.issue  travelPrice
11990 Log: IS.private.plan.openForMe  (true#'( ')#')
12000 Log: IS.private.plan.thePlan  [ findout( howTravel usr ) findout( departure usr
) findout( destination usr ) findout( travelMonth usr ) findout( class usr
) consultDB( travelPrice ) ]
12010 Log: IS.shared.com  [ destination # london ]
12020 Log: max-qud   o( asset:[ usr ] issue:howTravel oset:[ usr ] spkr:tSys status:open )
12030 Log: IS.shared.qud  [ o( asset:[ usr ] issue:howTravel oset:[ usr ] spkr:tSys status:open
) ]
12040 Log: IS.shared.lu  o( moves:[ move( ask howTravel ) ] spkr:tSys )
12050 Log: IS.shared.pu  o( moves:[ move( ask travelPrice ) ] spkr:usr )
12050 Log: IS:    ==END IS==
12110 Log: Generate:   o(contents:howTravel move:ask speaker:tSys)#' removed.'
12150 Event: startSpeaking   o(speaker:usr)
12260 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
12350 Event: utterance   o(contents:'By train.' speaker:usr)
12360 Log: close addressing status for issues owned by   tSys
12360 Trans: iAmSpeaking:<s> => [stopSpeaking] => iAmSilent:<s>
12360 Trans: questionsAsked:<s> => [stopSpeaking] => noQuestions:<s>
12370 Trans: othersTRPComingUp:<s> => othersTRP:<s>
12380 Event: closedTRP.here   o(speaker:tSys)
12400 Log: IS:    --BEGIN IS--
12410 Log: Registered DPs  [ hSys usr tSys ]
12420 Log: IS.private.agenda  [ findout( howTravel usr ) ]
12430 Log: IS.private.bel    [ ]
12440 Log: IS.private.plan.issue  travelPrice
12450 Log: IS.private.plan.openForMe  (true#'( ')#')
12460 Log: IS.private.plan.thePlan  [ findout( howTravel usr ) findout( departure usr
) findout( destination usr ) findout( travelMonth usr ) findout( class usr
) consultDB( travelPrice ) ]
12460 Log: IS.shared.com  [ destination # london ]
12530 Log: max-qud   o( asset:[ usr ] issue:howTravel oset:[ usr ] spkr:tSys status:closed
)
12540 Log: IS.shared.qud  [ o( asset:[ usr ] issue:howTravel oset:[ usr ] spkr:tSys status:closed
) ]
12550 Log: IS.shared.lu  o( moves:[ move( ask howTravel ) ] spkr:tSys )
12550 Log: IS.shared.pu  o( moves:[ move( ask travelPrice ) ] spkr:usr )
12560 Log: IS:    ==END IS==
12630 Event: openTRP.free.comingUp   o(speaker:usr)
12630 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
12630 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
12700 Event: says   o(contents:train move:answer speaker:usr)
12830 Event: doSelect      o
12940 Event: printIS      o
12950 Log: =usr=      answer:train
12980 Event: integrated    o(contents:train move:answer speaker:usr)
12990 Log: TransTaken: integrateAnswerAccept
12990 Trans: integrate:<s> => [says] => downdateQUD:<s>
13130 Log: TransTaken: downdateQUDDefault
13130 Trans: downdateQUD:<s> => loadPlan:<s>
13180 Log: TransTaken: loadPlanDefault
13180 Trans: loadPlan:<s> => execPlan:<s>
13240 Log: TransTaken: removeFindout

```

```

13250 Log: Removed:   findout(howTravel usr)
13380 Log: TransTaken: removeFindoutAgenda
13390 Log: Removed:   findout(howTravel usr)
13460 Log: TransTaken: execPlanDefault
13460 Trans: execPlan:<s> => update:<s>
13510 Trans: select.idle:<s> => [doSelect] => select.do:<s>
13510 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
13700 Event: printIS   o
13700 Log: Added action to agenda:   findout(departure usr)
13700 Trans: select.selectAction:<s> => select.selectMove:<s>
13760 Event: generate.generateMove   o(addressee:usr contents:howTravel#train move:icm#(acc#pos))
13770 Log: TransTaken: selectPosAcceptanceICM
13960 Event: generate.generateMove   o(addressee:usr contents:departure move:ask)
13970 Log: TransTaken: selectTask
14170 Event: stopSpeaking   o(speaker:usr)
14170 Trans: select.selectMove:<s> => select.idle:<s>
14220 Log: IS:   --BEGIN IS--
14230 Log: Registered DPs   [ hSys usr tSys ]
14240 Log: IS.private.agenda   [ ]
14250 Log: IS.private.bel   [ ]
14260 Log: IS.private.plan.issue   travelPrice
14260 Log: IS.private.plan.openForMe   (true#'( )#')
14270 Log: IS.private.plan.thePlan   [ findout( departure usr ) findout( destination
14270 Log: user ) findout( travelMonth usr ) findout( class usr ) consultDB( travelPrice
14270 Log: ) ]
14280 Log: IS.shared.com   [ howTravel # train destination # london ]
14290 Log: max-qud   o( aset:[ tSys ] issue:accept( howTravel # train ) issueObject:o(
14290 Log: aset:[ user ] issue:howTravel oset:[ ] spkr:tSys status:closed ) oset:[ ] spkr:usr
14290 Log: status:closed )
14300 Log: IS.shared.qud   [ o( aset:[ tSys ] issue:accept( howTravel # train ) issueObject:o(
14300 Log: aset:[ user ] issue:howTravel oset:[ ] spkr:tSys status:closed ) oset:[ ] spkr:usr
14300 Log: status:closed ) o( aset:[ user ] issue:howTravel oset:[ ] spkr:tSys status:closed ) ]
14310 Log: IS.shared.lu   o( moves:[ move( answer train ) ] spkr:usr )
14320 Log: IS.shared.pu   o( moves:[ move( ask howTravel ) ] spkr:tSys )
14330 Log: IS:   ==END IS==
14470 Log: IS:   --BEGIN IS--
14480 Log: IS.private.agenda   [ ]
14480 Log: Registered DPs   [ hSys usr tSys ]
14490 Log: IS.private.bel   [ ]
14500 Log: IS.private.plan.issue   travelPrice
14510 Event: selectNoreply.do   o
14510 Log: IS.private.plan.openForMe   (true#'( )#')
14520 Log: IS.private.plan.thePlan   [ findout( departure usr ) findout( destination
14520 Log: user ) findout( travelMonth usr ) findout( class usr ) consultDB( travelPrice
14520 Log: ) ]
14530 Log: IS.shared.com   [ howTravel # train destination # london ]
14540 Log: max-qud   o( aset:[ tSys ] issue:accept( howTravel # train ) issueObject:o(
14540 Log: aset:[ user ] issue:howTravel oset:[ ] spkr:tSys status:closed ) oset:[ ] spkr:usr
14540 Log: status:closed )
14560 Log: IS.shared.qud   [ o( aset:[ tSys ] issue:accept( howTravel # train ) issueObject:o(
14560 Log: aset:[ user ] issue:howTravel oset:[ ] spkr:tSys status:closed ) oset:[ ] spkr:usr
14560 Log: status:closed ) o( aset:[ user ] issue:howTravel oset:[ ] spkr:tSys status:closed ) ]
14570 Log: IS.shared.lu   o( moves:[ move( answer train ) ] spkr:usr )
14580 Log: IS.shared.pu   o( moves:[ move( ask howTravel ) ] spkr:tSys )
14590 Log: IS:   ==END IS==
14720 Trans: selectNoreply.active:<s> => [generate.generateMove] => selectNoreply.idle:<s>
14740 Trans: nothingToSay:<s> => wantToSpeak:<s>
14820 Trans: noQuestions:<s> => [generate.generateMove] => questions:<s>
14910 Log: close addressing status for issues owned by   user
14990 Trans: speaking:<s> => silence:<s>
15000 Trans: freeTRPComingUp:<s> => freeTRP:<s>
15010 Event: openTRP.free.here   o(speaker:usr)
15010 Trans: wantToSpeak:<s> => doGenerate:<s>
15020 Event: startSpeaking   o(speaker:tSys)
15030 Event: generateNext   o
15040 Event: doSelect   o
15080 Trans: iAmSilent:<s> => [startSpeaking] => iAmSpeaking:<s>
15080 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
15090 Event: openTRP.free.comingUp   o(speaker:tSys)
15150 Event: utterance   o(contents:'Yes, by train.' speaker:tSys)
15160 Event: says   o(contents:howTravel#train move:icm#(acc#pos) speaker:tSys)
15180 Event: addressing   o(addressee:usr speaker:tSys)
15260 Trans: select.idle:<s> => [doSelect] => select.do:<s>
15260 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
15360 Event: generateNext   o
15390 Event: printIS   o
15390 Log: Added action to agenda:   findout(departure usr)
15390 Trans: select.selectAction:<s> => select.selectMove:<s>
15570 Trans: select.selectMove:<s> => select.idle:<s>
15590 Event: doSelect   o
15750 Event: printIS   o
15820 Log: =tSys= icmaccpos:howTraveltrain
15840 Event: integrated   o(contents:howTravel#train move:icm#(acc#pos) speaker:tSys)
15860 Log: TransTaken: integratePosAcceptICM
15860 Trans: integrate:<s> => [says] => downdateQUD:<s>
15900 Log: TransTaken: downdateQUD
15910 Log: removed:   o( aset:[ user ] issue:howTravel oset:[ ] spkr:tSys status:closed )
16010 Log: TransTaken: downdateQUDDefault

```

```

16010 Trans: downdateQUD:<s> => loadPlan:<s>
16060 Log: TransTaken: loadPlanDefault
16060 Trans: loadPlan:<s> => execPlan:<s>
16190 Log: TransTaken: execPlanDefault
16190 Trans: execPlan:<s> => update:<s>
16230 Log: update addr for issues owned by tSys
16240 Trans: freeTRPComingUp:<s> => [addressing] => othersTRPComingUp:<s>
16250 Event: closedTRP.comingup o(speaker:tSys)
16270 Event: selectNoreply.do o
16310 Event: utterance o(contents:'What is the city of departure?' speaker:tSys)
16320 Event: says o(contents:departure move:ask speaker:tSys)
16340 Event: addressing o(addressee:usr speaker:tSys)
16420 Log: IS: --BEGIN IS--
16430 Log: Registered DPs [ hSys usr tSys ]
16440 Log: IS.private.agenda [ findout( departure usr ) ]
16450 Log: IS.private.bel []
16460 Log: IS.private.plan.issue travelPrice
16460 Log: IS.private.plan.openForMe (true#'( ')#')
16470 Log: IS.private.plan.thePlan [ findout( departure usr ) findout( destination
16480 Log: IS.shared.com [ howTravel # train destination # london ]
16490 Log: max-qud o( asset:[ tSys ] issue:accept( howTravel # train ) issueObject:o(
16490 Log: asset:[ usr ] issue:howTravel oset:[ ] spkr:tSys status:closed ) oset:[ hSys
16490 Log: tSys ] spkr:usr status:closed )
16500 Log: IS.shared.qud [ o( asset:[ tSys ] issue:accept( howTravel # train ) issueObject:o(
16500 Log: asset:[ usr ] issue:howTravel oset:[ ] spkr:tSys status:closed ) oset:[ hSys
16500 Log: tSys ] spkr:usr status:closed )
16510 Log: IS.shared.lu o( moves:[ move( icm # accpos howTravel # train ) ] spkr:tSys )
16520 Event: generateNext o
16520 Log: IS.shared.pu o( moves:[ move( answer train ) ] spkr:usr )
16520 Log: IS: ==END IS ==
16550 Trans: select.idle:<s> => [doSelect] => select.do:<s>
16720 Log: Added action to agenda: findout(departure usr)
16720 Trans: select.selectAction:<s> => select.selectMove:<s>
16730 Event: printIS o
16850 Trans: select.selectMove:<s> => select.idle:<s>
16880 Log: IS: --BEGIN IS--
16880 Log: Registered DPs [ hSys usr tSys ]
16960 Log: IS.private.agenda [ findout( departure usr ) ]
16970 Log: IS.private.bel []
16970 Log: IS.private.plan.issue travelPrice
16980 Log: IS.private.plan.openForMe (true#'( ')#')
16990 Log: IS.private.plan.thePlan [ findout( departure usr ) findout( destination
16990 Log: usr ) findout( travelMonth usr ) findout( class usr ) consultDB( travelPrice
16990 Log: ) ]
17000 Log: IS.shared.com [ howTravel # train destination # london ]
17010 Log: max-qud o( asset:[ tSys ] issue:accept( howTravel # train ) issueObject:o(
17010 Log: asset:[ usr ] issue:howTravel oset:[ ] spkr:tSys status:closed ) oset:[ hSys
17010 Log: tSys ] spkr:usr status:closed )
17020 Log: IS.shared.qud [ o( asset:[ tSys ] issue:accept( howTravel # train ) issueObject:o(
17020 Log: asset:[ usr ] issue:howTravel oset:[ ] spkr:tSys status:closed ) oset:[ hSys
17020 Log: tSys ] spkr:usr status:closed )
17030 Log: IS.shared.lu o( moves:[ move( icm # accpos howTravel # train ) ] spkr:tSys )
17040 Log: IS.shared.pu o( moves:[ move( answer train ) ] spkr:usr )
17040 Log: IS: ==END IS ==
17090 Log: Generate: o(contents:howTravel#train move:icm#(acc#pos) speaker:tSys)#
17090 Log: removed.'
17130 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
17270 Event: printIS o
17280 Log: =tSys= ask:departure
17300 Event: integrated o(contents:departure move:ask speaker:tSys)
17310 Log: TransTaken: integrateMeAsk
17320 Event: driver.speak o(contents:paris move:answer speaker:usr)
17320 Trans: integrate:<s> => [says] => downdateQUD:<s>
17320 Trans: questions:<s> => [says] => questionsAsked:<s>
17390 Log: TransTaken: downdateQUDDefault
17390 Trans: downdateQUD:<s> => loadPlan:<s>
17430 Log: TransTaken: loadPlanDefault
17430 Trans: loadPlan:<s> => execPlan:<s>
17560 Log: TransTaken: execPlanDefault
17560 Trans: execPlan:<s> => update:<s>
17600 Log: update addr for issues owned by tSys
17650 Event: stopSpeaking o(speaker:tSys)
17650 Trans: doGenerate:<s> => [generateNext] => nothingToSay:<s>
17690 Log: IS: --BEGIN IS--
17700 Log: Registered DPs [ hSys usr tSys ]
17760 Log: IS.private.agenda [ findout( departure usr ) ]
17770 Log: IS.private.bel []
17780 Log: IS.private.plan.issue travelPrice
17790 Log: IS.private.plan.openForMe (true#'( ')#')
17800 Log: IS.private.plan.thePlan [ findout( departure usr ) findout( destination
17800 Log: usr ) findout( travelMonth usr ) findout( class usr ) consultDB( travelPrice
17800 Log: ) ]
17810 Log: IS.shared.com [ howTravel # train destination # london ]
17820 Log: max-qud o( asset:[ usr ] issue:departure oset:[ usr ] spkr:tSys status:open )
17830 Log: IS.shared.qud [ o( asset:[ usr ] issue:departure oset:[ usr ] spkr:tSys status:open
17830 Log: ) o( asset:[ tSys ] issue:accept( howTravel # train ) issueObject:o( asset:[


```

```

17840 Log: usr ] issue:howTravel oset:[ hSys tSys status:closed ) oset:[ hSys tSys ] spkr:usr status:closed ) ]
IS.shared.lu   o( moves:[ move( ask departure ) move( icm # accpos howTravel
# train ) ] spkr:tSys )
17850 Log: IS.shared.pu   o( moves:[ move( answer train ) ] spkr:usr )
17850 Log: IS:    ==END IS==
17890 Log: IS:    --BEGIN IS--
17900 Log: Registered DPs [ hSys usr tSys ]
17910 Log: IS.private.agenda [ findout( departure usr ) ]
17920 Log: IS.private.bel [ ]
17930 Log: IS.private.plan.issue travelPrice
17940 Log: IS.private.plan.openForMe (true#'( ')#')
17950 Log: IS.private.plan.thePlan [ findout( departure usr ) findout( destination
usr ) findout( travelMonth usr ) findout( class usr ) consultDB( travelPrice
) ]
17950 Log: IS.shared.com [ howTravel # train destination # london ]
17960 Log: max-qud   o( asset:[ usr ] issue:departure oset:[ usr ] spkr:tSys status:open )
18030 Log: IS.shared.qud [ o( asset:[ usr ] issue:departure oset:[ usr ] spkr:tSys status:open
) o( asset:[ tSys ] issue:accept( howTravel # train ) issueObject:o( asset:[
usr ] issue:howTravel oset:[ ] spkr:tSys status:closed ) oset:[ hSys tSys ] spkr:usr status:closed ) ]
18040 Log: IS.shared.lu   o( moves:[ move( ask departure ) move( icm # accpos howTravel
# train ) ] spkr:tSys )
18050 Log: IS.shared.pu   o( moves:[ move( answer train ) ] spkr:usr )
18050 Log: IS:    ==END IS==
18110 Log: Generate: o(contents:departure move:ask speaker:tSys)#' removed.'
18150 Event: startSpeaking o(speaker:usr)
18310 Log: close addressing status for issues owned by tSys
18310 Trans: iamSpeaking:<s> => [stopSpeaking] => iamSilent:<s>
18310 Trans: questionsAsked:<s> => [stopSpeaking] => noQuestions:<s>
18320 Trans: othersTRPComingUp:<s> => othersTRP:<s>
18330 Event: closedTRP.here o(speaker:tSys)
18350 Event: utterance o(contents:Paris.' speaker:usr)
18400 Event: openTRP.free.comingUp o(speaker:usr)
18400 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
18400 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
18490 Event: says o(contents:paris move:answer speaker:usr)
18590 Event: doSelect   o
18710 Event: printIS   o
18720 Log: =usr=   answer:paris
18750 Event: integrated   o(contents:paris move:answer speaker:usr)
18750 Log: TransTaken: integrateAnswerAccept
18760 Trans: integrate:<s> => [says] => downdateQUD:<s>
18900 Log: TransTaken: downdateQUDAcceptance
18910 Log: removed: o(asset: issue:accept(howTravel#train) issueObject:o(asset: issue:howTravel
oset: spkr:tSys status:closed) oset: spkr:usr status:closed)
19000 Log: TransTaken: downdateQUDDefault
19000 Trans: downdateQUD:<s> => loadPlan:<s>
19050 Log: TransTaken: loadPlanDefault
19050 Trans: loadPlan:<s> => execPlan:<s>
19160 Log: TransTaken: removeFindout
19170 Log: Removed: findout(departure usr)
19230 Log: TransTaken: removeFindout
19240 Log: Removed: findout(destination usr)
19310 Log: TransTaken: removeFindoutAgenda
19320 Log: Removed: findout(departure usr)
19450 Log: TransTaken: execPlanDefault
19450 Trans: execPlan:<s> => update:<s>
19500 Trans: select.idle:<s> => [doSelect] => select.do:<s>
19500 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
19640 Event: printIS   o
19640 Log: Added action to agenda: findout(travelMonth usr)
19640 Trans: select.selectAction:<s> => select.selectMove:<s>
19760 Event: generate.generateMove o(addressee:usr contents:departure#paris move:icm#(acc#pos))
19780 Log: TransTaken: selectPosAcceptanceICM
19910 Event: generate.generateMove o(addressee:usr contents:travelMonth move:ask)
19920 Log: TransTaken: selectTask
20120 Trans: select.selectMove:<s> => select.idle:<s>
20170 Event: stopSpeaking o(speaker:usr)
20170 Log: IS:    --BEGIN IS--
20180 Log: Registered DPs [ hSys usr tSys ]
20190 Log: IS.private.agenda [ ]
20200 Log: IS.private.bel [ ]
20210 Log: IS.private.plan.issue travelPrice
20220 Log: IS.private.plan.openForMe (true#'( ')#')
20280 Log: IS.private.plan.thePlan [ findout( travelMonth usr ) findout( class usr
) consultDB( travelPrice ) ]
20290 Log: IS.shared.com [ departure # paris howTravel # train destination # london ]
20300 Log: max-qud   o( asset:[ tSys ] issue:accept( departure # paris ) issueObject:o(
asset:[ usr ] issue:departure oset:[ ] spkr:tSys status:closed ) oset:[ ] spkr:usr
status:closed )
20310 Log: IS.shared.qud [ o( asset:[ tSys ] issue:accept( departure # paris ) issueObject:o(
asset:[ usr ] issue:departure oset:[ ] spkr:tSys status:closed ) oset:[ ] spkr:usr
status:closed ) o( asset:[ usr ] issue:departure oset:[ ] spkr:tSys status:closed ) ]
20320 Log: IS.shared.lu   o( moves:[ move( answer paris ) ] spkr:usr )
20330 Log: IS.shared.pu   o( moves:[ move( ask departure ) move( icm # accpos howTravel
# train ) ] spkr:tSys )
20340 Log: IS:    ==END IS==
20420 Log: IS:    --BEGIN IS--
20430 Log: Registered DPs [ hSys usr tSys ]

```

```

20440 Log: IS.private.agenda      [ ]
20450 Log: IS.private.bel        [ ]
20460 Log: IS.private.plan.issue travelPrice
20470 Log: IS.private.plan.openForMe '(true#'( ')#')
20480 Log: IS.private.plan.thePlan [ findout( travelMonth usr ) findout( class usr
) consultDB( travelPrice ) ]
20480 Log: IS.shared.com      [ departure # paris howTravel # train destination # london ]
20550 Event: selectNoreply.do o(
20550 Log: max-qud   o( aset:[ tSys ] issue:accept( departure # paris ) issueObject:o(
20550 Log: aset:[ usr ] issue:departure oset:[ ] spkr:tSys status:closed ) oset:[ ] spkr:usr
status:closed )
20560 Log: IS.shared.qud      [ o( aset:[ tSys ] issue:accept( departure # paris ) issueObject:o(
20560 Log: aset:[ usr ] issue:departure oset:[ ] spkr:tSys status:closed ) oset:[ ] spkr:usr
status:closed ) o( aset:[ usr ] issue:departure oset:[ ] spkr:tSys status:closed ) ]
20570 Log: IS.shared.lu      o( moves:[ move( answer paris ) ] spkr:usr )
20580 Log: IS.shared.pu      o( moves:[ move( ask departure ) move( icm # accpos howTravel
# train ) ] spkr:tSys )
20590 Log: IS:    ==END IS==
20660 Trans: selectNoreply.active:<s> => [generate.generateMove] => selectNoreply.idle:<s>
20680 Trans: nothingToSay:<s> => wantToSpeak:<s>
20750 Trans: noQuestions:<s> => [generate.generateMove] => questions:<s>
20900 Log: close addressing status for issues owned by     usr
20930 Trans: speaking:<s> => silence:<s>
20940 Event: openTRP.free.here o(speaker:usr)
20940 Trans: freeTRPComingUp:<s> => freeTRP:<s>
20950 Trans: wantToSpeak:<s> => doGenerate:<s>
20960 Event: generateNext   o
20960 Event: startSpeaking   o(speaker:tSys)
20970 Event: doSelect   o
21020 Event: openTRP.free.comingUp   o(speaker:tSys)
21020 Trans: iAmSilent:<s> => [startSpeaking] => iAmSpeaking:<s>
21020 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
21140 Event: utterance   o(contents:'Yes, Paris.' speaker:tSys)
21150 Event: says   o(contents:departure#paris move:icm#(acc#pos) speaker:tSys)
21170 Event: addressing   o(addressee:usr speaker:tSys)
21200 Trans: select.idle:<s> => [doSelect] => select.do:<s>
21200 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
21320 Log: Added action to agenda:    findout(travelMonth usr)
21320 Trans: select.selectAction:<s> => select.selectMove:<s>
21330 Event: printIS   o
21350 Event: generateNext   o
21500 Trans: select.selectMove:<s> => select.idle:<s>
21530 Event: doSelect   o
21740 Event: printIS   o
21750 Log: =tSys= icmacpos:departureparis
21780 Event: integrated   o(contents:departure#paris move:icm#(acc#pos) speaker:tSys)
21780 Log: TransTaken: integratePosAcceptICM
21780 Trans: integrate:<s> => [says] => downdateQUD:<s>
21830 Log: TransTaken: downdateQUD
21830 Log: removed:   o( aset:[ usr ] issue:departure oset:[ ] spkr:tSys status:closed )
21910 Log: TransTaken: downdateQUDDefault
21910 Trans: downdateQUD:<s> => loadPlan:<s>
21950 Log: TransTaken: loadPlanDefault
21950 Trans: loadPlan:<s> => execPlan:<s>
22070 Log: TransTaken: execPlanDefault
22070 Trans: execPlan:<s> => update:<s>
22110 Log: update addr for issues owned by     tSys
22130 Trans: freeTRPComingUp:<s> => [addressing] => othersTRPComingUp:<s>
22140 Event: closedTRP.comingup   o(speaker:tSys)
22170 Log: IS:    --BEGIN IS--
22180 Log: Registered DPs [ hSys usr tSys ]
22190 Log: IS.private.agenda      [ findout( travelMonth usr ) ]
22200 Event: selectNoreply.do o(
22200 Log: IS.private.bel        [ ]
22210 Log: IS.private.plan.issue travelPrice
22210 Log: IS.private.plan.openForMe '(true#'( ')#')
22280 Log: IS.private.plan.thePlan [ findout( travelMonth usr ) findout( class usr
) consultDB( travelPrice ) ]
22290 Log: IS.shared.com      [ departure # paris howTravel # train destination # london ]
22300 Log: max-qud   o( aset:[ tSys ] issue:accept( departure # paris ) issueObject:o(
22300 Log: aset:[ usr ] issue:departure oset:[ ] spkr:tSys status:closed ) oset:[ hSys
tSys ] spkr:usr status:closed )
22310 Log: IS.shared.qud      [ o( aset:[ tSys ] issue:accept( departure # paris ) issueObject:o(
22310 Log: aset:[ usr ] issue:departure oset:[ ] spkr:tSys status:closed ) oset:[ hSys
tSys ] spkr:usr status:closed ) ]
22320 Log: IS.shared.lu      o( moves:[ move( icm # accpos departure # paris ) ] spkr:tSys )
22320 Log: IS.shared.pu      o( moves:[ move( answer paris ) ] spkr:usr )
22330 Log: IS:    ==END IS==
22380 Event: utterance   o(contents:'What month would you like to travel?' speaker:tSys)
22390 Event: says   o(contents:travelMonth move:ask speaker:tSys)
22410 Event: addressing   o(addressee:usr speaker:tSys)
22440 Trans: select.idle:<s> => [doSelect] => select.do:<s>
22590 Event: generateNext   o
22620 Event: printIS   o
22620 Log: Added action to agenda:    findout(travelMonth usr)
22620 Trans: select.selectAction:<s> => select.selectMove:<s>
22740 Trans: select.selectMove:<s> => select.idle:<s>
22770 Log: IS:    --BEGIN IS--

```

```

22780 Log: Registered DPs [ hSys usr tSys ]
22790 Log: IS.private.agenda [ findout( travelMonth usr ) ]
22790 Log: IS.private.bel []
22860 Log: IS.private.plan.issue travelPrice
22870 Log: IS.private.plan.openForMe (true#'( ')#')
22880 Log: IS.private.plan.thePlan [ findout( travelMonth usr ) findout( class usr )
) consultDB( travelPrice ) ]
22890 Log: IS.shared.com [ departure # paris howTravel # train destination # london ]
22900 Log: IS.shared.qud [ o( asset:[ tSys ] issue:accept( departure # paris ) issueObject:o(
aset:[ usr ] issue:departure oset:[ ] spkr:tSys status:closed ) oset:[ hSys
tSys ] spkr:usr status:closed ) ]
22900 Log: max-qud o( asset:[ tSys ] issue:accept( departure # paris ) issueObject:o(
aset:[ usr ] issue:departure oset:[ ] spkr:tSys status:closed ) oset:[ hSys
tSys ] spkr:usr status:closed )
22910 Log: IS.shared.lu o( moves:[ move( icmp # accpos departure # paris ) ] spkr:tSys )
22920 Log: IS.shared.pu o( moves:[ move( answer paris ) ] spkr:usr )
22930 Log: IS: ==END IS==
22970 Log: Generate: o(contents:departure#paris move:icmp#(acc#pos) speaker:tSys)#
removed.'
23010 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
23170 Event: printIS o
23180 Log: =tSys= ask:travelMonth
23200 Event: integrated o(contents:travelMonth move:ask speaker:tSys)
23210 Log: TransTaken: integrateMeAsk
23220 Event: driver.speak o(contents:april move:answer speaker:usr)
23220 Trans: integrate:<s> => [says] => downdateQUD:<s>
23220 Trans: questions:<s> => [says] => questionsAsked:<s>
23290 Log: TransTaken: downdateQUDDefault
23290 Trans: downdateQUD:<s> => loadPlan:<s>
23330 Log: TransTaken: loadPlanDefault
23330 Trans: loadPlan:<s> => execPlan:<s>
23450 Log: TransTaken: execPlanDefault
23450 Trans: execPlan:<s> => update:<s>
23490 Log: update addr for issues owned by tSys
23540 Event: stopSpeaking o(speaker:tSys)
23540 Trans: doGenerate:<s> => [generateNext] => nothingToSay:<s>
23580 Log: IS: --BEGIN IS--
23590 Log: Registered DPs [ hSys usr tSys ]
23600 Log: IS.private.agenda [ findout( travelMonth usr ) ]
23610 Log: IS.private.bel []
23620 Log: IS.private.plan.issue travelPrice
23630 Log: IS.private.plan.openForMe (true#'( ')#')
23630 Log: IS.private.plan.thePlan [ findout( travelMonth usr ) findout( class usr
) consultDB( travelPrice ) ]
23640 Log: IS.shared.com [ departure # paris howTravel # train destination # london ]
23650 Log: max-qud o( asset:[ usr ] issue:travelMonth oset:[ usr ] spkr:tSys status:open
)
23720 Log: IS.shared.qud [ o( asset:[ usr ] issue:travelMonth oset:[ usr ] spkr:tSys
status:open ) o( asset:[ tSys ] issue:accept( departure # paris ) issueObject:o(
aset:[ usr ] issue:departure oset:[ ] spkr:tSys status:closed ) oset:[ hSys tSys ] spkr:usr status:closed ) ]
23730 Log: IS.shared.lu o( moves:[ move( ask travelMonth ) move( icmp # accpos departure
# paris ) ] spkr:tSys )
23740 Log: IS.shared.pu o( moves:[ move( answer paris ) ] spkr:usr )
23740 Log: IS: ==END IS==
23780 Log: IS: --BEGIN IS--
23790 Log: Registered DPs [ hSys usr tSys ]
23800 Log: IS.private.agenda [ findout( travelMonth usr ) ]
23810 Log: IS.private.bel []
23820 Log: IS.private.plan.issue travelPrice
23830 Log: IS.private.plan.openForMe (true#'( ')#')
23830 Log: IS.private.plan.thePlan [ findout( travelMonth usr ) findout( class usr
) consultDB( travelPrice ) ]
23840 Log: IS.shared.com [ departure # paris howTravel # train destination # london ]
23850 Log: max-qud o( asset:[ usr ] issue:travelMonth oset:[ usr ] spkr:tSys status:open
)
23860 Log: IS.shared.qud [ o( asset:[ usr ] issue:travelMonth oset:[ usr ] spkr:tSys
status:open ) o( asset:[ tSys ] issue:accept( departure # paris ) issueObject:o(
aset:[ usr ] issue:departure oset:[ ] spkr:tSys status:closed ) oset:[ hSys tSys ] spkr:usr status:closed ) ]
23870 Log: IS.shared.lu o( moves:[ move( ask travelMonth ) move( icmp # accpos departure
# paris ) ] spkr:tSys )
23880 Log: IS.shared.pu o( moves:[ move( answer paris ) ] spkr:usr )
23890 Log: IS: ==END IS==
24000 Log: Generate: o(contents:travelMonth move:ask speaker:tSys)#
removed.'
24040 Event: startSpeaking o(speaker:usr)
24150 Log: close addressing status for issues owned by tSys
24150 Trans: iAmSpeaking:<s> => [stopSpeaking] => iamSilent:<s>
24150 Trans: questionsAsked:<s> => [stopSpeaking] => noQuestions:<s>
24160 Trans: othersTRPComingUp:<s> => othersTRP:<s>
24170 Event: closedTRP.here o(speaker:tSys)
24280 Event: utterance o(contents:'In April.' speaker:usr)
24290 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
24290 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
24300 Event: openTRP.free.comingUp o(speaker:usr)
24380 Event: says o(contents:april move:answer speaker:usr)
24430 Event: doSelect o
24600 Event: printIS o
24610 Log: =usr= answer:april
24640 Event: integrated o(contents:april move:answer speaker:usr)

```

```

24650 Log: TransTaken: integrateAnswerAccept
24650 Trans: integrate:<s> => [says] => downdateQUD:<s>
24740 Log: TransTaken: downdateQUDAcceptance
24740 Log: removed: o(asset: issue:accept(departure#paris) issueObject:o(asset: issue:departure
oset: spkr:tSys status:closed) oset: spkr:usr status:closed)
24900 Log: TransTaken: downdateQUDDefault
24900 Trans: downdateQUD:<s> => loadPlan:<s>
24950 Log: TransTaken: loadPlanDefault
24950 Trans: loadPlan:<s> => execPlan:<s>
25000 Log: TransTaken: removeFindout
25010 Log: Removed: findout(travelMonth usr)
25080 Log: TransTaken: removeFindoutAgenda
25090 Log: Removed: findout(travelMonth usr)
25230 Log: TransTaken: execPlanDefault
25230 Trans: execPlan:<s> => update:<s>
25270 Trans: select.idle:<s> => [doSelect] => select.do:<s>
25270 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
25470 Event: printIS o
25470 Log: Added action to agenda: findout('class' usr)
25470 Trans: select.selectAction:<s> => select.selectMove:<s>
25530 Event: generate.generateMove o(addressee:usr contents:travelMonth#april move:icm#(acc#pos))
25540 Log: TransTaken: selectPosAcceptanceICM
25680 Event: generate.generateMove o(addressee:usr contents:'class' move:ask)
25690 Log: TransTaken: selectAsk
25880 Trans: select.selectMove:<s> => select.idle:<s>
25940 Log: IS: --BEGIN IS--
25940 Log: Registered DPs [ hSys usr tSys ]
25950 Log: IS.private.agenda []
25960 Log: IS.private.bel []
25970 Log: IS.private.plan.issue travelPrice
25980 Log: IS.private.plan.openForMe (true#( ')#')
25990 Log: IS.private.plan.thePlan [ findout( class usr ) consultDB( travelPrice ) ]
26050 Log: IS.shared.com [ travelMonth # april departure # paris howTravel # train
destination # london ]
26060 Event: stopSpeaking o(speaker:usr)
26060 Log: max-qud o( asset:[ tSys ] issue:accept( travelMonth # april ) issueObject:o(
asset:[ usr ] issue:travelMonth oset:[ ] spkr:tSys status:closed ) oset:[ ]
spkr:usr status:closed )
26070 Log: IS.shared.qud [ o( asset:[ tSys ] issue:accept( travelMonth # april ) issueObject:o(
asset:[ usr ] issue:travelMonth oset:[ ] spkr:tSys status:closed ) oset:[ ]
spkr:usr status:closed ) o( asset:[ usr ] issue:travelMonth oset:[ ] spkr:tSys status:closed ) ]
26080 Log: IS.shared.lu o( moves:[ move( answer april ) ] spkr:usr )
26090 Log: IS.shared.pu o( moves:[ move( ask travelMonth ) move( icm # accpos departure
# paris ) ] spkr:tSys )
26100 Log: IS: --END IS--
26180 Log: IS: --BEGIN IS--
26190 Log: Registered DPs [ hSys usr tSys ]
26200 Log: IS.private.agenda []
26210 Log: IS.private.bel []
26220 Log: IS.private.plan.issue travelPrice
26220 Log: IS.private.plan.openForMe (true#( ')#')
26230 Log: IS.private.plan.thePlan [ findout( class usr ) consultDB( travelPrice ) ]
26240 Log: IS.shared.com [ travelMonth # april departure # paris howTravel # train
destination # london ]
26250 Log: max-qud o( asset:[ tSys ] issue:accept( travelMonth # april ) issueObject:o(
asset:[ usr ] issue:travelMonth oset:[ ] spkr:tSys status:closed ) oset:[ ]
spkr:usr status:closed )
26260 Log: IS.shared.qud [ o( asset:[ tSys ] issue:accept( travelMonth # april ) issueObject:o(
asset:[ usr ] issue:travelMonth oset:[ ] spkr:tSys status:closed ) oset:[ ]
spkr:usr status:closed ) o( asset:[ usr ] issue:travelMonth oset:[ ] spkr:tSys status:closed ) ]
26270 Log: IS.shared.lu o( moves:[ move( answer april ) ] spkr:usr )
26330 Event: selectNoreply.do o
26340 Log: IS.shared.pu o( moves:[ move( ask travelMonth ) move( icm # accpos departure
# paris ) ] spkr:tSys )
26340 Log: IS: --END IS--
26410 Trans: selectNoreply.active:<s> => [generate.generateMove] => selectNoreply.idle:<s>
26430 Trans: nothingToSay:<s> => wantToSpeak:<s>
26500 Trans: noQuestions:<s> => [generate.generateMove] => questions:<s>
26650 Log: close addressing status for issues owned by usr
26680 Trans: speaking:<s> => silence:<s>
26690 Trans: freeTRPComingUp:<s> => freeTRP:<s>
26700 Event: openTRP.free.here o(speaker:usr)
26700 Trans: wantToSpeak:<s> => doGenerate:<s>
26710 Event: generateNext o
26710 Event: startSpeaking o(speaker:tSys)
26720 Event: doSelect o
26770 Trans: iamSilent:<s> => [startSpeaking] => iAmSpeaking:<s>
26770 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>
26780 Event: openTRP.free.comingUp o(speaker:tSys)
26830 Event: utterance o(contents:'Yes, in April.' speaker:tSys)
26900 Event: says o(contents:travelMonth#april move:icm#(acc#pos) speaker:tSys)
26920 Event: addressing o(addressee:usr speaker:tSys)
26950 Trans: select.idle:<s> => [doSelect] => select.do:<s>
26950 Trans: selectNoreply.idle:<s> => [doSelect] => selectNoreply.active:<s>
27070 Log: Added action to agenda: findout('class' usr)
27070 Trans: select.selectAction:<s> => select.selectMove:<s>
27080 Event: printIS o
27110 Event: generateNext o

```

```

27250 Trans: select.selectMove:<s> => select.idle:<s>
27280 Event: doSelect      o
27490 Event: printIS      o
27500 Log:   =tSys= icmaccpos:travelMonthapril
27530 Event: integrated    o(contents:travelMonth#april move:icm#(acc#pos) speaker:tSys)
27530 Log:   TransTaken: integratePosAcceptICM
27530 Trans: integrate:<s> => [says] => downdateQUD:<s>
27580 Log:   TransTaken: downdateQUD
27580 Log:   removed: o( asset:[ usr ] issue:travelMonth oset:[ ] spkr:tSys status:closed )
27660 Log:   TransTaken: downdateQUDDefault
27660 Trans: downdateQUD:<s> => loadPlan:<s>
27700 Log:   TransTaken: loadPlanDefault
27700 Trans: loadPlan:<s> => execPlan:<s>
27830 Log:   TransTaken: execPlanDefault
27830 Trans: execPlan:<s> => update:<s>
27870 Log:   update addr for issues owned by tSys
27880 Trans: freeTRPComingUp:<s> => [addressing] => othersTRPComingUp:<s>
27890 Event: closedTRP.comingup o(speaker:tSys)
27930 Log:   IS: --BEGIN IS--
27930 Log:   Registered DPs [ hSys usr tSys ]
27940 Log:   IS.private.agenda [ findout( class usr ) ]
27950 Event: selectNoreply.do o
27950 Log:   IS.private.bel [ ]
27960 Log:   IS.private.plan.issue travelPrice
27970 Log:   IS.private.plan.openForMe (true#'( ')#')
27980 Log:   IS.private.plan.thePlan [ findout( class usr ) consultDB( travelPrice ) ]
27990 Log:   IS.shared.com [ travelMonth # april departure # paris howTravel # train
destination # london ]
27990 Log:   max-qud o( asset:[ tSys ] issue:accept( travelMonth # april ) issueObject:o(
asset:[ usr ] issue:travelMonth oset:[ ] spkr:tSys status:closed ) oset:[ hSys
tSys ] spkr:usr status:closed )
28000 Log:   IS.shared.qud [ o( asset:[ tSys ] issue:accept( travelMonth # april ) issueObject:o(
asset:[ usr ] issue:travelMonth oset:[ ] spkr:tSys status:closed ) oset:[ hSys
tSys ] spkr:usr status:closed ) ]
28070 Log:   IS.shared.lu o( moves:[ move( icm # accpos travelMonth # april ) ] spkr:tSys
)
28080 Log:   IS.shared.pu o( moves:[ move( answer april ) ] spkr:usr )
28090 Log:   IS: ==END IS==
28140 Event: utterance o(contents:'What class did you have in mind?' speaker:tSys)
28150 Event: says o(contents:'class' move:ask speaker:tSys)
28170 Event: addressing o(addressee:usr speaker:tSys)
28190 Trans: select.idle:<s> => [doSelect] => select.do:<s>
28360 Event: generateNext o
28370 Log:   Added action to agenda: findout('class' usr)
28370 Trans: select.selectAction:<s> => select.selectMove:<s>
28380 Event: printIS o
28490 Trans: select.selectMove:<s> => select.idle:<s>
28520 Log:   IS: --BEGIN IS--
28530 Log:   Registered DPs [ hSys usr tSys ]
28540 Log:   IS.private.agenda [ findout( class usr ) ]
28550 Log:   IS.private.bel [ ]
28560 Log:   IS.private.plan.issue travelPrice
28570 Log:   IS.private.plan.openForMe (true#'( ')#')
28580 Log:   IS.private.plan.thePlan [ findout( class usr ) consultDB( travelPrice ) ]
28580 Log:   IS.shared.com [ travelMonth # april departure # paris howTravel # train
destination # london ]
28650 Log:   max-qud o( asset:[ tSys ] issue:accept( travelMonth # april ) issueObject:o(
asset:[ usr ] issue:travelMonth oset:[ ] spkr:tSys status:closed ) oset:[ hSys
tSys ] spkr:usr status:closed )
28660 Log:   IS.shared.qud [ o( asset:[ tSys ] issue:accept( travelMonth # april ) issueObject:o(
asset:[ usr ] issue:travelMonth oset:[ ] spkr:tSys status:closed ) oset:[ hSys
tSys ] spkr:usr status:closed ) ]
28670 Log:   IS.shared.lu o( moves:[ move( icm # accpos travelMonth # april ) ] spkr:tSys
)
28680 Log:   IS.shared.pu o( moves:[ move( answer april ) ] spkr:usr )
28690 Log:   IS: ==END IS==
28730 Log:   Generate: o(contents:travelMonth#april move:icm#(acc#pos) speaker:tSys)#
removed.
28770 Trans: selectNoreply.active:<s> => [selectNoreply.do] => selectNoreply.idle:<s>
28930 Event: printIS o
28940 Log:   =tSys= ask:class
28960 Event: integrated o(contents:'class' move:ask speaker:tSys)
28970 Log:   TransTaken: integrateMeAsk
28980 Event: driver.speak o(contents:business move:answer speaker:usr)
28980 Trans: integrate:<s> => [says] => downdateQUD:<s>
28980 Trans: questions:<s> => [says] => questionsAsked:<s>
29060 Log:   TransTaken: downdateQUDDefault
29060 Trans: downdateQUD:<s> => loadPlan:<s>
29090 Log:   TransTaken: loadPlanDefault
29090 Trans: loadPlan:<s> => execPlan:<s>
29160 Log:   TransTaken: execPlanDefault
29160 Trans: execPlan:<s> => update:<s>
29260 Log:   update addr for issues owned by tSys
29300 Event: stopSpeaking o(speaker:tSys)
29310 Trans: doGenerate:<s> => [generateNext] => nothingToSay:<s>
29350 Log:   IS: --BEGIN IS--
29360 Log:   IS.private.agenda [ findout( class usr ) ]
29360 Log:   Registered DPs [ hSys usr tSys ]

```

```

29370 Log: IS.private.bel      [ ]
29380 Log: IS.private.plan.issue   travelPrice
29390 Log: IS.private.plan.openForMe   (true?'( ')#')
29400 Log: IS.private.plan.thePlan   [ findout( class usr ) consultDB( travelPrice ) ]
29410 Log: IS.shared.com   [ travelMonth # april departure # paris howTravel # train
destination # london ]
29420 Log: max-qud   o( asset:[ usr ] issue:class oset:[ usr ] spkr:tSys status:open )
29430 Log: IS.shared.lu   o( moves:[ move( ask class ) move( icm # accpos travelMonth
# april ) ] spkr:tSys )
29430 Log: IS.shared.qud   [ o( asset:[ usr ] issue:class oset:[ usr ] spkr:tSys status:open
) o( asset:[ tSys ] issue:accept( travelMonth # april ) issueObject:o( asset:[
usr ] issue:travelMonth oset:[ ] spkr:tSys status:closed ) oset:[ hSys tSys ] spkr:usr status:closed ) ]
29440 Log: IS.shared.pu   o( moves:[ move( answer april ) ] spkr:usr )
29510 Log: IS: ==END IS==
29550 Log: IS: --BEGIN IS--
29560 Log: IS.private.agenda   [ findout( class usr ) ]
29560 Log: Registered DPs   [ hSys usr tSys ]
29570 Log: IS.private.bel   [ ]
29580 Log: IS.private.plan.issue   travelPrice
29590 Log: IS.private.plan.openForMe   (true?'( ')#')
29600 Log: IS.private.plan.thePlan   [ findout( class usr ) consultDB( travelPrice ) ]
29610 Log: IS.shared.com   [ travelMonth # april departure # paris howTravel # train
destination # london ]
29620 Log: max-qud   o( asset:[ usr ] issue:class oset:[ usr ] spkr:tSys status:open )
29630 Log: IS.shared.lu   o( moves:[ move( ask class ) move( icm # accpos travelMonth
# april ) ] spkr:tSys )
29630 Log: IS.shared.qud   [ o( asset:[ usr ] issue:class oset:[ usr ] spkr:tSys status:open
) o( asset:[ tSys ] issue:accept( travelMonth # april ) issueObject:o( asset:[
usr ] issue:travelMonth oset:[ ] spkr:tSys status:closed ) oset:[ hSys tSys ] spkr:usr status:closed ) ]
29640 Log: IS.shared.pu   o( moves:[ move( answer april ) ] spkr:usr )
29650 Log: IS: ==END IS==
29700 Log: Generate: o(contents:'class' move:ask speaker:tSys) '#' removed.
29810 Event: startSpeaking   o(speaker:usr)
29910 Log: close addressing status for issues owned by   tSys
29910 Trans: iAmSpeaking:<s> => [stopSpeaking] => iamSilent:<s>
29910 Trans: questionsAsked:<s> => [stopSpeaking] => noQuestions:<s>
29920 Trans: othersTRPComingUp:<s> => othersTRP:<s>
29930 Event: closedTRP.here   o(speaker:tSys)
30000 Trans: outsideChart:<s> => [startSpeaking] => speaking:<s>
30000 Trans: trpChart:<s> => [startSpeaking] => freeTRPComingUp:<s>

```