D3.1: Extended Information State Modeling

Ivana Kruijff-Korbayová (editor),
Gabriel Amores, Nate Blaylock, Stina Ericsson,
Guillermo Pérez, Kalliroi Georgila, Michael Kaisser,
Staffan Larsson, Oliver Lemon, Pilar Manchón, Jan Schehl

Distribution: Public

TALK
Talk and Look: Tools for Ambient Linguistic Knowledge
IST-507802 Deliverable 3.1

February 10, 2006

Project funded by the European Community
under the Sixth Framework Programme for
Research and Technological Development

The deliverable identification sheet is to be found on the reverse of this page.

Project ref. no.	IST-507802
Project acronym	TALK
Project full title	Talk and Look: Tools for Ambient Linguistic Knowledge
Instrument	STREP
Thematic Priority	Information Society Technologies
Start date / duration	01 January 2004 / 36 Months

Security	Public
Contractual date of delivery	M24 = December 2005
Actual date of delivery	February 10, 2006
Deliverable number	3.1
Deliverable title	D3.1: Extended Information State Modeling
Type	Report
Status & version	Final version February 10, 2006
Number of pages	87 (excluding front matter)
Contributing WP	3
WP/Task responsible	USAAR
Other contributors	DFKI, UGOT, USE, UEDIN
Author(s)	Ivana Kruijff-Korbayová (editor), Gabriel Amores, Nate Blaylock, Stina Ericsson, Guillermo Pérez, Kalliroi Georgila, Michael Kaisser, Staffan Larsson, Oliver Lemon, Pilar Manchón, Jan Schehl
EC Project Officer	Evangelia Markidou (Anne Bajart)
Keywords	information state, context, multimodality, collaborative problem solving, information structure

The partners in TALK are:

Saarland University	USAAR
University of Edinburgh HCRC	UEDIN
University of Gothenburg	UGOT
University of Cambridge	UCAM
University of Seville	USE
Deutches Forschungszentrum fur Künstliche Intelligenz	DFKI
Linguamatics	LING
BMW Forschung und Technik GmbH	BMW
Robert Bosch GmbH	BOSCH

For copies of reports, updates on project activities and other TALK-related information, contact:

The TALK Project Co-ordinator
Prof. Manfred Pinkal
Computerlinguistik
Fachrichtung 4.7 Allgemeine Linguistik
Postfach 15 11 50
66041 Saarbrücken, Germany
pinkal@coli.uni-sb.de
Phone +49 (681) 302-4343 - Fax +49 (681) 302-4351

Copies of reports and other material can also be accessed via the project's administration homepage, `http://www.talk-project.org`

Contents

Summary

This deliverable reports on the work carried out in Task 3.1, as part of WP3. Task 3.1 contributes to the overall objective of WP3 to support flexible and adaptive system output presentation in multiple modes by investigating and subsequently implementing the extensions that are needed in the information state representations and the corresponding information state updating in the respective showcases.

We present a range of information state extensions with respect to the systems prior to TALK. The main common denominator of the extensions developed in the various showcases has been to handle multimodal input and output. We have developed extended representations of the multimodal discourse context, allowing us to keep track of the available modalities and their capacity to convey information at any given point, as well as other contextual features helping decisions concerning multimodal fission and the realization of linguistic output. In addition, we have developed extensions needed with respect to specific research issues pursued within the different showcases. This for example involves extended task representations reflecting the collaborative problem solving state, and extensions with additional dialogue context features needed for the purpose of reinforcement learning.

Chapter 1

Introduction

The aim of WP3 is to explore the use of the information state for advanced multimodal presentation of system output, which is aimed to facilitate easy and efficient interaction, adapted to the dialogue context, the situation, the user, the available modalities, etc. The basic use made of the information state (IS) in the Information State Update-based (ISU) framework is the following: When a conversation participant hears an utterance, she computes an update from her current information state to a new information state. Next, she computes an appropriate dialogue contribution on the basis of the new information state (and updates again). The ISU approach uses abstract representations for dialogue states and update rules which allow the generic characterization of flexible dialogue strategies. This enables the same code for dialogue management techniques to be used for different natural languages and for different domains. The IS can be read and updated by several different modules which access precisely the information that they need. This enables a modular architecture which allows generic solutions for dialogue technology.

Prior to the TALK project, research in the ISU approach explored the uses of the shared knowledge and commitments accumulated in the IS, for example, to help interpretation or provide immediate feedback of the system's understanding. The output produced by the ISU-based system(s) had commonly been a straightforward realization of the underlying propositional content of a dialogue move, typically obtained by template-based generation. However, in a dialogue with a flexible interaction flow, output generation needs to become more flexible too. Previous work addressed some aspects of contextually appropriate system output realization in the spoken modality, in particular intonation. But there still remains a lot that can be done. For example, referring expressions need to be constructed to ensure proper identification of referents w.r.t. contextually available competitors, and generally the linguistic structure of the entire utterance (i.e., syntactic constructions, word order, intonation) needs to properly reflect its information structure w.r.t. the current context.

The use of multiple modalities adds new possibilities, because system output can be distributed in various ways. For example, multimodality enables graphical presentation of various kinds of feedback (e.g., what the system has understood) or background information (e.g., the list of available options about which further dialogue may be carried out), while reserving the speech channel for information that advances the interaction (e.g. making a suggestion or asking a question). Up to our knowledge, these possibilities have not been systematically investigated and utilized in dialogue systems.

It is the goal of Task 3.1 to design and maintain *extended* information states in the various showcases, containing detailed and structured representations of the dialogue context, in order to support advanced multimodal dialogue and system output presentation in various phases of a dialogue. Extensions refer to

information state representations used by the partners prior to TALK. We are concerned with various kinds of extensions, both to improve dialogue behavior (e.g., clarification requests or cooperative responses) and to support advanced multimodal presentation (e.g., context-dependent multimodal fission or contextually appropriate realization of spoken output).

We build on the use of the information state to record the accumulation of shared commitments and to monitor the question(s) under discussion, and extend it with additional structured information to achieve more elaborate dialogue context modeling that has commonly been done so far in the ISU framework. The extended information state aims to provide a representation of what the dialogue is about at various stages, and how the subsequent stages relate to one another. This does not necessarily mean keeping a complete transcript of the interchange; rather, we have in mind structured representations of the information that is retained beyond its local context. Having the globally relevant information retained in the extended information state opens new possibilities for user-adaptivity in the ISU approach. The extended information state (or its parts) can be used either in subsequent interactions with the same user or in interactions with other users with similar needs.

In particular, our objective in Task 3.1 has been to develop the extended information state by (i) defining what information should be retained in the extended information state beyond local context, (ii) developing representations of the extended information state in a structured way useful for dialogue management, output presentation and input interpretation, and (iii) developing methods to maintain the extended information state during a dialogue.

Extensions of Information State Representations in the TALK Showcases The ISU-based systems developed prior to TALK mostly only used verbal communication. In all the TALK showcases we developed extended IS representations to support the extension of interaction modes from purely verbal (written or spoken) to multimodal (combining verbal and graphical, where what is displayed can be either text, e.g., tables or lists, or images, e.g., maps). This includes in particular an explicit representation of the available modalities, their capacity to convey content, and keeping track of which content has been conveyed in which modality. All these aspects play a role in system output planning.

Furthermore, we have developed extensions towards more elaborate task- and utterance content representations, with the purpose to improve the support for flexible and adaptive dialogue. The extensions addressed in the various showcases reflect the research foci of the respective partners.

In the SAMMIE system, we wanted to be able to handle more fine-grained information about the state of the task the system and the user are jointly working on to support, e.g., the representation of multiple problem solutions or the assignment of meta-level status information to task information (such as, e.g., accepted, rejected, postponed, abandoned, etc.). To address these needs, we have designed and implemented a formalism based on the theory of dialogue as collaborative problem solving, thus improving the modeling of the task part of the IS, which generally has not received much attention in the ISU-based systems so far.

Another aspect of flexible and adaptive dialogue has to do with “context-awareness” in interpretation and generation. In WP3, we are particularly interested in contextually appropriate realization of system turns with respect to properly distinguishing between information that is assumed to already be known to the user and new information, in order to make the interaction more efficient and natural. This involves, e.g., handling anaphoric expressions, ellipsis, word order, intonation, and also features of graphical display (cf. Task 3.2). However, in order to support decisions about the context-dependent aspects of surface realization, we need to represent the relevant information in the information state in a suitable way. Developing

IS extensions with these concerns in mind has been the a research focus for both the SAMMIE and GODIS systems. We have developed more fine-grained representation of discourse history which is used for both multimodal fission and to determine context-dependent aspects of linguistic realization, such as referring expressions and information structure.

Finally, UEDIN has been concerned with extending the IS representations in order to support reinforcement learning (RL). This involves history features (so that learning has access to dialogue history), task features (so that learning can take account of task completion), and rewards (so that the RL algorithm has an appropriate reward signal).

Report Outline The deliverable is structured as follows. In Chapter 2 provides a brief introduction to the Information State Update-based paradigm of dialogue modeling. We survey the information state representations in several ISU-based dialogue systems in other projects prior to TALK. We also mention available toolkits for developing ISU-based systems. The subsequent chapters of the deliverable discuss the information state extensions in the respective TALK showcases: SAMMIE in Chapter 3, GODIS in Chapter 4, the UEDIN/UCAM system in Chapter 5 and Delfos in Chapter 6. In every chapter we motivate the extensions, describe the extended information state and the update rules that maintain it. We close with a final discussion in Chapter 7. The appendix contais brief descriptions of the software prototypes accompanying this deliverable. They are preliminary implementations of the respective showcases, which implement the information state extensions discussed in this deliverable.

Chapter 2

Information State-Based Modeling

In this chapter we briefly introduce the Information State Update-based paradigm of dialogue modeling. We survey the information state representations in several pre-TALK ISU-based dialogue systems, which illustrate the range of different approaches to ISU-based dialogue modeling. It is clear that although they share the ISU-based approach, there are differences in what each dialogue model consists of and how it is built as a dialogue proceeds. This is not surprising, given that there exists no uniform standard theory of dialogue. The information state representations of the systems developed in TALK will be described in detail in the following chapters, where we present the extensions required in the various systems and application scenarios, in order to support simultaneous multimodal presentation as well as to increase flexibility and adaptivity of the modeled dialogues.

2.1 The ISU-Based Approach to Dialogue Modeling

The general notion *Information State* (IS) refers to the information which is necessary to distinguish one dialogue state from all other dialogue states. In this sense the IS can be understood in a similar way as *conversational score*, *discourse context* or *mental state*.

The basic idea of the Information State Update-based approach to dialogue modeling is that each dialogue participant maintains an information state that is updated with each utterance, and serves to compute an appropriate next dialogue contribution. This approach thus provides a framework for formalizing information states in human/computer dialogue systems. Dialogue systems following this approach work on a data structure (the Information State), which is modified by the update rules, most of which are related to dialogue moves.

More precisely, any theory of dialogue modeling following the ISU-based approach should account for the following components [TBC+99]:

- A description of the *informational components* of the information state.
- A *formal representation* of these components.
- A set of *dialogue moves* triggering the update of the information state.
- A set of *update rules*, needed to govern the updating of the information state.

- A *control strategy* to decide which update rule(s) to select a given point in the dialog.

Below we overview the state of the art in developing ISU-based dialogue systems at the beginning of the TALK project. We first briefly review available toolkit for implementing such systems, and then we summarize the main features of the information states implemented in a range of ISU-based systems. This overview shows that information state representations differ from system to system, depending on the adopted theory of dialogue as well as on the particular research concerns pursued in various projects.

2.2 Software Tools for Building ISU-Based Dialogue Managers

To support the implementation of ISU-based systems, the following toolkits have been developed:[1]

TrindiKit [LBGK02][2] – The development is managed by the Gothenburg University Dialogue Systems Lab.

DIPPER [BKLO03a][3] – Developed by the Language Technology Group at the University of Edinburgh and CSLI Stanford, its Dialogue Move Engine builds on TrindiKit.

MIDIKI [BDG+03][4] – developed at The MITRE Corporation.

Rubin [FB03] – developed at CLT Sprachtechnologie, Saarbrücken.

They provide a generic shell together with a language to implement the core of a dialogue manager (among others, the structure of the IS and the IS update rules) and the interfaces between modules.

The above systems have been specially crafted as ISU dialogue management engines. As further outlined in Status Report D5.3s2 "In-Car Showcase based on TALK Libraries" and in Chapter 3 below, the SAMMIE system (USAAR/DFKI) uses the tool PATE [Kem04, Pfl04] for its dialogue management engine. PATE is a general production-rule system loosely based on ACT-R, and was *not* designed specifically to support ISU dialogue management. However, we have been effectively able to use it to support ISU dialog. This opens up the possibility in the ISU field of utilizing work on more general production-rule systems.

2.3 Systems Using the ISU-Based Approach

As indicated already, the ISU-based approach aims at providing a general framework for developing and testing discourse theories rather than presenting a single theory. In the last years several different theories and systems have been described with the notion of information state. We briefly overview several systems below.

[1] TrindiKit and DIPPER are open source tools currently being further developed in the TALK project.
[2] `http://www.ling.gu.se/projekt/trindi/trindikit/`
[3] `http://www.ltg.ed.ac.uk/dipper/`
[4] `http://midiki.sourceforge.net/`

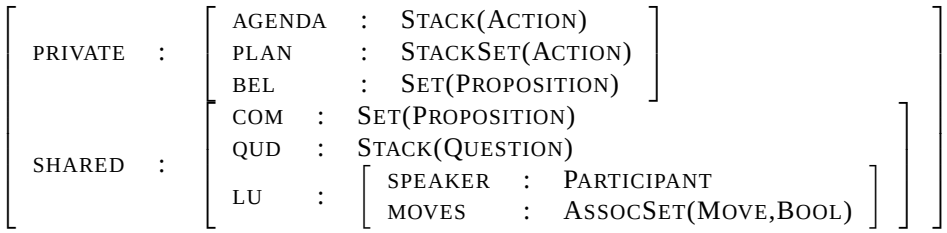

Figure 2.1: The "Canonical" Information State in GODIS

GODIS Developed at Gothenburg University, GODIS [Lar02] is a multi-lingual experimental dialogue system making use of TrindiKit. Before the start of the TALK project, implementations existed for the domains of a traveling service, an intelligent house, and to access a VCR.

The information state in GODIS is a version of the *dialogue gameboard* [Gin96a]. It is divided into a PRIVATE and a SHARED part, the former containing private information only known to the system, the latter containing information that the system assumes to be shared by the user (cf. Figure 2.1). Besides information about the latest utterance (speaker and move(s)), the SHARED part contains shared commitments (a set of propositions) and QUD (a stack of questions under discussion). When a question is asked, it is pushed onto the QUD, and is popped off when it is answered. The SHARED part is empty at the beginning of the dialogue and successively being filled during the progression of the dialog. With this method grounding phenomena are modeled. In the PRIVATE part, the plan contains the system's long-term goals, while the agenda contains more immediate actions.

GODIS distinguishes eight dialogue move types: *ask*, *answer*, *repeat* , *request_repeat*, *greet*, *goodbye*, *thank* and *quit*. It uses quite simple algorithms for control, update and selection modules, as well as keyword-based interpretation of user utterances and template-based generation of system output.

GODIS implements two variants of a grounding strategy, optimistic and cautious grounding. In the latter the system will always ask for a confirmation of its understanding before it regards some information as being grounded and before it will finally be written in the shared part of the information state.

Slightly different information states have been used at different stages in the development of GODIS as part of various applications in different projects, including Trindi, D'Homme and Siridus. Figure 2.1 shows the canonical IS version. The current version of the IS used in the TALK project is described in detail in Chapter 4.

EDIS Developed at University of Edinburgh within the Trindi project, EDIS [MPT00] divides the information state into three parts: A common ground, a semi-public part and a private part (see Figure 2.2). The common ground part contains four types of information: obligations of dialogue participants to perform actions (OBL), social commitments that participants have that propositions hold (SCP), a dialogue history containing acts that have been performed already (DH) and conditional statements to establish either obligations or commitments (COND). The semi-public part contains discourse units that are grounded (DUs). The private part contains intentions of the system (INT).

MIDAS Also developed in the Trindi project, MIDAS [BG00] uses the discourse representation structures of DRT as its underlying semantic framework. Figure 2.3 gives an example of an information state used in MIDAS. The system makes use of theorem provers for dialogue act interpretation and to perform conditional tests on the update rules.

$$\begin{bmatrix} \text{G} & : & PT\text{-}R \\ \text{CDU} & : & \begin{bmatrix} \text{C} & : & PT\text{-}R \\ \text{ID} & : & \text{DU-ID} \end{bmatrix} \\ \text{PDU} & : & \begin{bmatrix} \text{C} & : & PT\text{-}R \\ \text{ID} & : & \text{DU-ID} \end{bmatrix} \\ \text{UDUs} & : & \text{List(DU-ID)} \\ \text{INT} & : & \text{List(Action)} \end{bmatrix} \qquad PT\text{-}R : \begin{bmatrix} \text{DH} & : & \text{List(Action)} \\ \text{OBL} & : & \text{List(Action)} \\ \text{SCP} & : & \text{List(Prop)} \\ \text{COND} & : & \text{List(Action)} \end{bmatrix}$$

Figure 2.2: Information State in EDIS

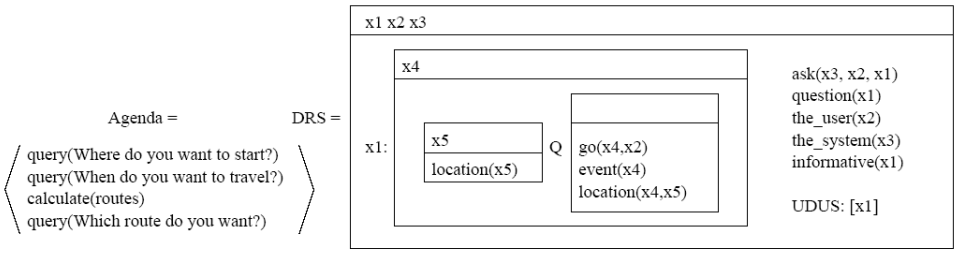

Figure 2.3: Information State in MIDAS

Autoroute Another TrindiKit-based system developed in the Siridus project remodels **Autoroute** [Lew98] with the information state notion. It formalizes conversational games as recursive transition networks. Figure 2.4 gives an example of the top-level structure of the information state, which is a record. A dialogue participant's role as rational agent is distinguished from that of a conversational game-player. The elements *ACTION* and *AGENDAITEM*, which can be seen in the figure, are themselves shortcuts for other records.

$$\begin{bmatrix} \textsc{rational\_agent} & : & \begin{bmatrix} \textsc{plan} & : & \textsc{Stack(Action)} \\ \textsc{scoreboard} & : & \textsc{Set(Prop)} \end{bmatrix} \\ \textsc{game\_player} & : & \begin{bmatrix} \textsc{agendaitem} & : & \textsc{Stack(Agendaitem)} \\ \textsc{currtoken} & : & \textsc{Token} \\ \textsc{alltokens} & : & \textsc{Stack(Set(Prop))} \end{bmatrix} \end{bmatrix}$$

Figure 2.4: Information State in Autoroute

DELFOS Finally, also DELFOS, developed at the University of Seville, is based on the ISU approach. It has been improved and extended during the D'Homme and Siridus projects, and is currently being extended in TALK. DELFOS is especially designed to deal with Natural Command Languages (i.e., languages for controlling devices) and although a number of specific dialogue moves have been implemented for the current applications, more dialogue moves are currently under development to extend its functionality in

new applications. The IS in DELFOS is shown schematically in Figure 2.5. It consists of DMOVE, an application-independent dialogue move, TYPE, a specific application-dependent type of dialogue move, ARGS, additional information needed to complete the dialogue move and CONT, the actual content of the structures described above. The current version of the IS used in the TALK project is described in detail in Chapter 6.

$$\begin{bmatrix} \text{DMOVE} : & \ldots \\ \text{TYPE} : & \ldots \\ \text{ARGS} : & \ldots \\ \text{CONT} : & \ldots \end{bmatrix}$$

Figure 2.5: Information State in DELFOS

Most of the work mentioned above has taken place in several consecutive projects, TRINDI[5], D'Homme[6] and SIRIDUS[7]. The ISU-based approach to dialogue modeling has also been influential outside these projects. Here we mention ISU-based systems which are interesting, because they tackle dialogue modeling for quite challenging applications: intelligent tutoring (BEETLE) and dialogue with mobile robots (GODOT and WITAS).

BEETLE Developed at the University of Edinburgh, BEETLE [ZMC+03] is a dialogue-enhanced Basic Electricity and Electronics Tutorial Learning Environment. Beside modules for natural language understanding, generation and planning. It uses TIS, a dialogue manager based on TrindiKit.

BEETLE borrows large parts of its information state structure from EDIS. Many update rules were assumed unchanged; some were adapted for the purpose of tutoring and some new ones were created, mainly those which capture dialogue moves that are only present in the tutorial dialogue genre (for example *hinting* moves).

GODOT Developed also at the University of Edinburgh, GODOT [BKO03] is a mobile robot integrated with a spoken dialogue system, which the user can direct to specific locations, ask for information about its status, and supply information about its environment. Its information state consists of the gram- mar currently loaded by the speech recogniser, contact (whether or not there is communicative contact with someone), input (the results of speech recognition), nextmoves (the next dialogue moves to be realized by the robot), lastmoves (the latest dialogue moves produced by the user), and interpretation (consisting of the discourse representation structure of the ongoing dialogue and a first-order model generated for it). Similarly to MIDAS, GODOT uses inference engines for model building and theorem proving to assist in natural language interpretation and to test the applicability of dialogue update rules.

WITAS Developed at Stanford University, WITAS [LBGP01] is a dialogue system supporting multimodal activity-oriented dialogues. An activity in this sense is meant as a task than can be carried out by devices. In the case of WITAS the device was a semi-autonomous mobile robot helicopter which could

[5] http://www.ling.gu.se/projekt/trindi
[6] http://www.ling.gu.se/projekt/dhomme
[7] http://www.ling.gu.se/projekt/siridus/

perform activities as starting, landing, searching for an object, following a vehicle etc. The information state in WITAS consists of the following:

- *A Dialogue Move Tree* (DMT) containing a history of dialogue contributions. It is organized by threads and based on activities. It provides a model of how incoming utterances can be interpreted in the current dialogue context.
- *An Activity Tree* which contains a representation of current and planned activities and their execution status.
- *A System Agenda* which is a list of private issues which have yet to be raised.
- *A Pending List* storing questions the system has asked, but which the user has not answered so far.
- *Salience Groups* consisting of objects that have been referenced in the dialogue so far.
- *A Modality Buffer* which keeps track of mouse gestures until they are bound to user utterances or recognized as independent gesture expressions.

The context representation using the dialogue move tree and activity tree was designed to support multiple interleaved threads of dialogue about different activities and their execution status.

2.4 Summary

In this chapter, we gave a brief survey of several representatives of the ISU-based approach to dialogue modeling that existed before the start of the TALK project.

Although these systems all share the underlying concept of viewing dialogue utterances as update acts on an information state, we can see that there are differences among the systems in what each dialogue model consists of, how it is represented and how the representations are built as a dialogue proceeds. This reflects the state-of-the-art in dialogue modeling: there is no standard theory of dialogue. Which is hardly surprising given that dialogues involve complex phenomena which are not yet fully understood. Different projects also pursue different research goals, and in accordance with that often concentrate on different issues in their information state representations. In the TALK project, we aim to advance our understanding of what makes natural interactions, to provide the conceptual tools that enable us to model flexible and adaptive dialogue in the ISU-based framework, and extend the ISU-based paradigm towards multimodal interaction.

As we have also seen, the ISU-based systems developed so far mostly used only verbal communication (apart from the WITAS system). In TALK, we are extending the interaction modes from purely verbal (written or spoken) to multimodal (combining verbal and graphical, where what is displayed can be either text, e.g., tables or lists, or images, e.g., maps, and user input includes clicks).

In the next chapters we describe the information state representations of the systems developed in TALK in detail, focusing on modeling flexible and adaptive multimodal dialogue. We address a range of extensions of the information states of the systems that existed prior to TALK. The extensions investigated and adopted in the individual systems are overlapping in some aspects, and complementary in others, in accordance to the research foci pursued by the TALK partners.

Chapter 3

The SAMMIE Extended Information State

In this chapter,[1] we describe joint work between USAAR and DFKI on formulating the extended information state (EIS) for the SAMMIE dialogue system [BBG+05]. The focus of our extensions to typical information states reflects our research focuses in the project, which includes advanced multimodal planning and presentation for the rest of WP3, as well as flexible and portable ontology-driven dialogue management for WP2. Accordingly, the SAMMIE information state contains on the one hand extended representation of the multimodal context, keeping track of information communicated through speech and through text or graphics on the screen, and on the other hand extended task representations modeling the collaborative problem solving state.

The remainder of this chapter is as follows: in Section 3.1, we discuss the general information needed in our EIS; in Section 3.2, we then report on how this information is structured within the EIS; and in Section 3.3, we discuss how the EIS is updated during a dialogue.

3.1 Extended Information

To support our work in WP2 and WP3, we have identified several types of necessary information needed in the information state (IS). We discuss each below.

3.1.1 Contextual Information

In general, discourse specific information in monomodal dialogue systems is mainly used to fulfill two major tasks: The enrichment of context given by a single user utterance and the resolution of referring expressions. However, in multimodal systems there is additionally a need to keep track of the used input modalities (e.g, for resolution of crossmodal references) but also to have a proper representation of the capabilities of given output modalities (which might be static or dynamic, depending on the system context, e.g. one system that serves interaction possibilities through several different output devices) in order to create effective presentations of the system's discourse contributions. Furthermore, in some cases it might be interesting to have a proper presentation of information about the user's habits, expertise, and

[1] Portions of this chapter have been taken from [Bla05] and [BA05]

contextual preferences concerning the orientation towards a specific modality[2].

Hence a multimodal system's information state needs to have not only a representation of contextual information about discourse, but also a representation of modality-specific information and user-specific information which can be used to plan system output suited to a given context.

Discourse Information

As noted above, two major tasks concerning discourse processing are enrichment of given input by context and the resolution of referring expressions and ellipses. examples 1 and 2 present dialogue excerpts from a typical interaction with the SAMMIE system, exemplifying these tasks.

(1) U(User)1: Play Yesterday by the Beatles
S(System)1: From which of these albums:
'One', 'The Greatest Hits' or 'Red Album'
U2: From the album 'One'.

(2) U3: Show me all Beatles albums.
S2: I have these four Beatles albums.
[shows a list of the four album names]
U4: Play the third one.

In the first example the system needs to interpret the internal representation of the partial utterance U2 by enriching it through information from U1 in order to plan the next system action. In the second example, the user refers verbally to an object (U4) of a previously introduced collection (S2), which was presented graphically.

Discourse information is clearly also crucial for various decisions in the generation of natural language output, such as the generation of referring expressions or the determination of information structure (see also Chapter 4). Example 3 and 4 illustrates a possible difference in the choice of referring expression (definite NP vs. pronoun) depending on the preceding multimodal discourse context:

(3) U: Which songs are on the Red Album?
S: The Red Album contains these songs [shows a list of the songs]

(4) U: Show me the Beatles albums.
S: I have these four Beatles albums. [shows a list of the four album names]
U: Which songs are on this one? [selects the Red Album]
S: It contains these songs [shows a list of the songs]

In 3, the user initiates a new task and refers to an album for the first time by speech. The system repeats that referring expression for grounding purposes (to display what it understood). In 4, on the other hand, the system is displaying a small set of album names, and the user makes an unambiguous selection on the screen. Grounding feedback is not needed here, the system can safely pronominalize.

[2]There are various reasons for a user to focus on a specific modality, e.g, cognitive restrictions due to other ongoing tasks a user has beside system interaction (like car-driving) or (cognitive) disabilities that influence a user's interaction capabilities.

As these tasks need access to a multimodal contextual representation of the preceding discourse there is the need to have an appropriate model of the discourse history within the information state.

Modality-specific Information

As discussed above, previous work in ISU-based dialogue has typically been speech only. To represent the information state while using multiple possible input and output modalities, the following additional information is also needed:

Information About Available Modalities Information about a system's interaction capabilities in terms of given modality-channels is necessary for multimodal fission when speaking about multimodality, and speech recognition, input interpretation and natural language generation when speaking about multilinguality. As the structure of such information mostly relies on the modality specific resources a systems owns the reader is referred to D3.3, where these resources will be described in detail.

Modality Requests An interesting phenomenon of multimodal dialogue, which we term *modality requesting*, is that utterances can explicitly request that a response be presented in a specific modality. Compare, for example, the following possible utterances from the MP3 player domain:

(5) Which Beatles albums do you have?

(6) Tell me which Beatles albums you have.

(7) Show me which Beatles albums you have.

At the task level, we represent the semantics of these utterances the same — a request to identify a set of resources given a constraint (see CPS discussion below). However, (6) and (7) additionally include information about the requested modality for the response (speech and display, respectively) — something which we need to take into account in planning the system's next move.

Note that these modality requests need not explicitly accompany task-level requests. They can apparently also be used to request the re-presentation of the system turn in a different modality. Consider the following two dialogue sequences:

(8) U5: Which Beatles albums do you have?
S3: I have these four Beatles albums.
[shows a list of the four album names]
U6: Can you read them to me?

(9) U7: Which Beatles albums do you have?
S4: I have One, Number One Hits, Yellow Submarine, and Abbey Road.
U8: Show them to me on the screen.

In the first example, the user requests that the system re-present the resulting set of albums through the speech modality. In the second example, she requests the display modality. In both of these examples, the *content* of what is to be presented presumably does not change, just the modality. Modality request handling has not yet been implemented in the SAMMIE system, but we hope to include this in the final showcase version.

User-specific Information

When a user interacts with a multimodal system, there might be situations where communication is preferably limited to just some of these modalities. One typical situation in this case is, when system interaction becomes a secondary task (e.g., while driving a car). Hence, it is important to have a representation of the user's attention state for the system communication channels and/or a representation of the user's cognitive load due to other tasks she has to fulfill while interacting. This contextual information needs to be updated dynamically every time user attention state/cognitive load changes.

Furthermore, it seems plausible for us to have a representation of a user's expertise concerning system usage within the information state in order to adapt system responses to the level of interaction experience of a user.

For the SAMMIE system we modeled such user aspects as simple state variables, grouped as user-specific information in order to be able to provide user adaptive presentation planning. We refer to D3.2 where we will describe how multimodal presentation planning is influenced by these factors.

3.1.2 Planning and Execution Status

We are interested in building *conversational agents* — autonomous agents which can communicate with humans through natural language dialogue. In order to support dialogue with autonomous agents, we need to be able to model dialogue about the range of activities an agent may engage in, including goal selection, planning, execution, monitoring, replanning, and so forth.

Current models of dialogue are only able to support a small subset of these sorts of agent activities. Plan-based dialogue models, for example, typically model either planning dialogue (e.g., [Car90, GS90]) or execution dialogue (e.g., [CLNO91, LGP02]), but not both. Also, most plan-based dialogue models make the assumption that agents already have a high-level goal which they are pursuing.

More concretely, to support dialogue about both planning and execution, it is important that the information state be able to distinguish the various states of the task. For example, at a certain stage of the dialogue, a task may be planned but not yet executed; it may be planned and under execution; or it may have already been successfully executed. Below we describe how these and various other task states are recorded in the EIS.

3.1.3 State of the Decision-making Process

Another aspect of collaborative dialogue that has often been overlooked in typical information states is that of decision making. Most slot-filling dialogue systems, (e.g., [SP00, BR03]) allow the user to choose the value of a slot (e.g., a destination city) but do not support *collaboration* about which value the slot should take. The difference is illustrated in the following two dialogues:

(10) S: Which city would you like to fly to?
U: Miami.

(11) S: Which city would you like to fly to?
U: Something on the beach.
S: I have available tickets to Miami.
U: No, I went there last winter.
S: There are also tickets available to San Diego, Cancún, or Puerto Rico.
U: OK, San Diego.

The first dialogue is an example of a typical segment which can be easily handled by slot-filling dialogue models, in which the information state includes a slot for the destination city which is either empty or filled. We would argue, however, that such information is not sufficient to handle the second dialogue, where slot value is collaboratively decided. Here, instead of answering the question directly (e.g., by providing a city), the user places a *constraint* on possible slot values (i.e., that they must be close to a beach). The system uses this information to *suggest* a possible value, which the user *evaluates* and *rejects*. The system then provides several possible *alternatives* for the value, and the user finally chooses one.

Several researchers ([CCC00, DJTM00]) have shown that this type of exchange (which we will call *collaborative decision making*) occurs in many domains (including the travel booking domain). We also observe collaborative decision making in the MP3 domain:

(12) S: Create a new playlist
S: Which songs should I put on the playlist?
U: Lite my fire
S: I am looking for Lite my fire.
This song exists from various artists. Do you mean the one from The Doors?
U: No.
S: Should I show you the whole list?
U: Yes.
S: I am showing you the list. Which version should I add to the playlist?
U: Deep Purple.

Below we will describe how the SAMMIE IS is structured so that it can represent such collaborative decision-making dialogues. Particularly, we will show how we record information about arbitrary constraints on potential values, the identification of competing alternatives, and the evaluation and rejection or final acceptance of a value.

3.1.4 Grounding Status of Objects

For advanced multimodal presentation, it is necessary that the information state record not only what objects have been discussed, but also their grounding status [Tra94].

We will show below how our IS records not only which objects are in the common ground, but also the *information* about each object that is in the common ground — information which is vital in, among other things, planning linguistic (co)referring expressions, as well as in choosing which information to display about a particular object.

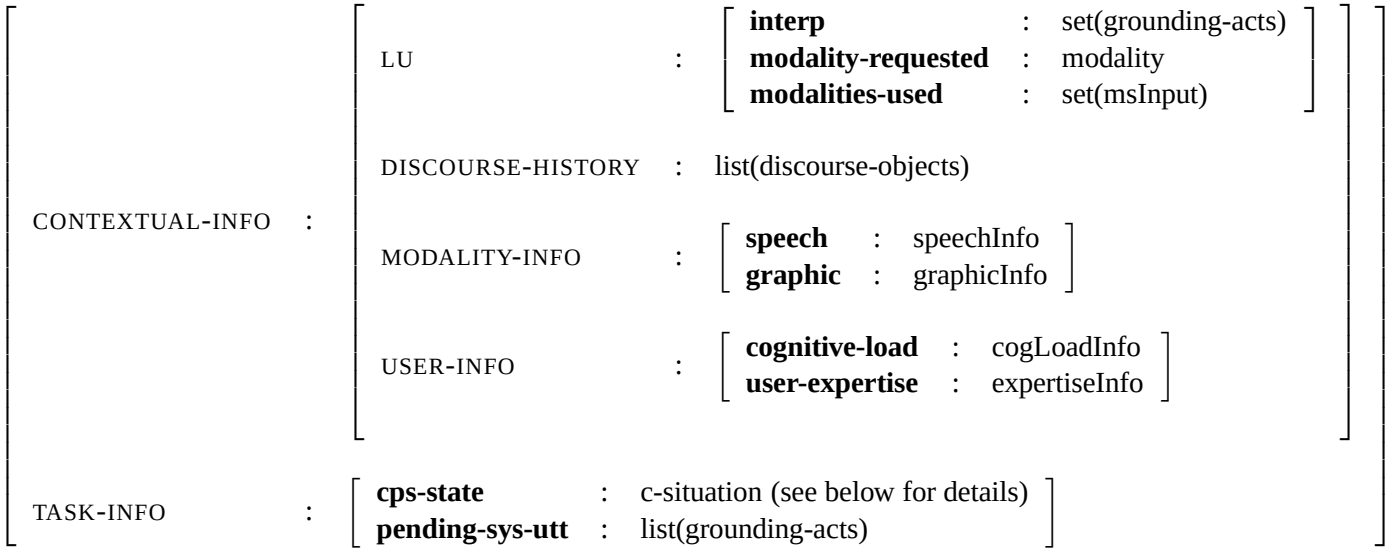

Figure 3.1: Structure of the SAMMIE Information State

3.2 Structuring the Information

In this section, we discuss how the information mentioned in the previous section is structured within the SAMMIE information state. As this deliverable is meant to be more than just a description of the actual system, we also include theoretical contributions which have not yet been implemented. Such cases are marked explicit as such in the text.

The overall information state (IS) of the SAMMIE system is shown in Figure 3.1. At a high level, the IS is divided into two different sections: CONTEXTUAL-INFO and TASK-INFO. This separation reflects the architecture and distribution of labor within our system. Our core system architecture (as described in Deliverable 5.2 [BBG$^+$05]) is very similar to that in the TRIPS system [AFS01, BAF02] which separates task and discourse-level processing. Unlike the GODIS system (described in Chapter 4), we do not explicitly separate private and shared information within the IS. This does not reflect a difference in view (we also believe this separation to be important). Rather, we have not included that level of detail as it is not vital to our research aims within this work package. Most information in our IS is considered shared, unless stated otherwise.

For the rest of this section, we detail these two areas of our IS. We first describe the structure of contextual information, and then task information.

3.2.1 Contextual Information

The contextual information partition of the IS is used to record important information at the discourse or linguistic level of the dialogue, but also modality- and user-specific information. The first part is necessary primarily for the processes of interpretation and generation, while information about the user and the capabilities of given modalities are important for context adaptive presentation planning.

The contextual information is divided into information about the last user utterance, capabilities of modalities a system can deal with, specific information about the user in order to adapt the presentation of

system output to her needs, and discourse history, which contains an extraction of the important features of previous turns which comprise the dialogue thus far.

Latest User Utterance The field *lu* records information about the latest user utterance which comprises the user input according to the used modalities, the completely interpreted user input (represented as a set of grounding acts), and if any, the modality that was explicitly requested.

Discourse History

The *discourse-history* captures the information that has been made public in the conversation, represented by a salient list of domain specific discourse objects, ordered by recency. Each introduced discourse object relies on two classes of information, modality specific information and domain information. Our approach to discourse representation follows the approach adopted in the SmartKom system [PAB03]; the discourse model employs the three-tiered context representation proposed in [Lup91] where the linguistic layer is generalized to a modality layer. The advantage of this approach to discourse representation lies in the unified representation of discourse objects introduced by different modalities. Note, that in the SAMMIE system we are currently using are more simplified version of the discourse history, as the implementation of the three-tiered context representation isn't finished yet.

Modality Information

The field *modality-info* records information about the available modalities, namely speech and graphics. Speech information currently describes the supported languages, ASR-, TTS-systems and information about graphics comprises of the display size and descriptions of different views on objects sets (e.g., map and table). As already stated, these information depend closely on modality specific resources and will be presented in more detail in D3.1.

User Information

The field *user-info* stores the field *cognitive-load* with possible state values *low*, *mid*, *high* or *extreme*, and the field *user-expertise* with possible state values *beginner* or *power-user*.

3.2.2 Task Information

Whereas the discourse information part of the IS details discourse information communicated in the different modalities, the task information section concerns itself only about the state of domain tasks. This task information is based on our model of collaborative problems solving [Bla05, BA05], which we describe below. We then discuss how the structure of this model allows us to represent the desired information we discussed in the previous section.

3.2.3 Collaborative Problem Solving

We see problem solving (PS) as the process by which a (single) agent chooses and pursues *objectives* (i.e., goals). Specifically, we model it as consisting of the following three general phases:

- *Determining Objectives*: In this phase, an agent manages objectives, deciding to which it is committed, which will drive its current behavior, etc.
- *Determining and Instantiating Recipes for Objectives*: In this phase, an agent determines and instantiates a recipe to use to work towards an objective. An agent may either choose a recipe from its recipe library, or it may choose to *create* a new recipe via planning.
- *Executing Recipes and Monitoring Success*: In this phase, an agent executes a recipe and monitors the execution to check for success.

There are several things to note about this general description. First, we do not impose any strict ordering on the phases above. For example, an agent may begin executing a partially-instantiated recipe and do more instantiation later as necessary. An agent may also adopt and pursue an objective in order to help it in deciding what recipe to use for another objective.

It is also important to note that our purpose here is not to specify a specific *problem-solving strategy* or prescriptive model of how an agent *should* perform problem solving. Instead, we want to provide a general descriptive model that enables agents with different PS strategies to still communicate.

Collaborative problem solving (CPS) follows a process similar to single-agent problem solving. Here two agents jointly choose and pursue objectives in the same stages (listed above) as single agents.

There are several things to note here. First, the level of collaboration in the problem solving may vary greatly. In some cases, for example, the collaboration may be primarily in the planning phase, but one agent will actually execute the plan alone. In other cases, the collaboration may be active in all stages, including the planning and execution of a joint plan, where both agents execute actions in a coordinated fashion. Again, we want a model that will cover the range of possible levels of collaboration.

3.2.4 Problem-Solving Objects

The basic building blocks of our formal CPS model are problem-solving (PS) objects, which we represent as typed feature structures. PS object types form a single-inheritance hierarchy, where children inherit or specialize features from parents. Instances of these types are then used in problem solving.[3]

In our CPS model, we define types for the upper level of an ontology of PS objects, which we term *abstract PS objects*. These abstract PS objects are used to model problem-solving at a domain-independent level, and all operators (discussed below) operate on them. The model is then specialized to a domain by inheriting and instantiating domain-specific types and instances from the PS objects. The operators, however, do not change with domain, which supports reasoning done at a domain-independent level.

We first describe the abstract PS objects and then how they are specialized.

Abstract PS Objects

The following are the six abstract PS objects from which all other domain-specific PS objects inherit:

Objective A goal, subgoal or action. For example, in a rescue domain, objectives could include rescuing a person, evacuating a city, and so forth; in the MP3 domain, objectives include playing an album,

[3] Due to space constraints, we omit here a discussion of the formal representation of objects. We refer readers to [Bla05] for details.

adding a song to a playlist, etc.. We consider objectives to be actions rather than states, allowing us to unify the concepts of action and goal.

Recipe Beliefs of how to attain an objective. A recipe library can be expanded or modified through (collaborative or single-agent) planning.

Constraint A restriction on an object. Constraints are used to restrict possible solutions in the problem-solving process as well as possible referents in object identification.

Evaluation An assessment of an object's value within a certain problem-solving context. Agents will often evaluate several options before choosing one.

Situation The state of the world (or a possible world). In all but the simplest domains, agents may only have partial knowledge about a given situation.

Resource All other objects in the domain. These include include real-world objects (e.g., airplanes, ambulances, songs in the MP3 database) as well as concepts (e.g., song titles, artist names)

In order to ensure that all objects in our problem-solving model are labeled with a unique ID, we introduce a basic type *object* with a single attribute ID. (Formal definitions for all objects discussed here are found in Figures 3.2 and 3.3.) This is then used as the root of all objects in the hierarchy.

Each of the abstract PS objects share a set of common features. We put these common features in a new type, *ps-object*, which is the common parent of all of the abstract PS objects. We briefly describe its features here and then continue by giving the type declarations for each of the abstract PS objects in turn.

ps-object inherits from *object* and therefore contains an ID attribute (not shown — we will typically not list inherited features). It also has one additional attribute: CONSTRAINTS. This provides a way of describing the *ps-object* with a set of *constraints*.

It is important to note that the type of the CONSTRAINTS attribute is not simply a set of type *constraint*. Rather, it is one of a special class of middleman types we call *slots*. As these middleman types are a vital part of the CPS model, we take a brief aside here to discuss them before continuing with the abstract PS objects.

Slots and Fillers Collaborative problem solving can be seen as a decision-making process with respect to choosing and pursuing objectives. In modeling problem solving, we want to model more than just the decisions made; we want to model the decision-making *process* itself.

Within our model, decisions can be seen as the choosing of values (objects) or sets of values for certain roles. For example, agents decide on a set of objectives to pursue; for each objective they have, agents must decide on a (single) recipe to use in pursuing it; and so forth. A straightforward way of modeling these decisions would be to include, for example, a feature RECIPE in an objective which takes a recipe value, and represents the current recipe the agents are using to pursue this objective. Similarly, we could define a feature OBJECTIVES at the top level which would hold the set of objectives the agents are currently committed to.

Doing this, however, would only model the agents' *decision*, but not the *process* the agents followed/are following in making that decision. In deciding on a recipe to use for an objective, agents may identify several possible recipes as possibilities and evaluate each one. They may similarly narrow down the space

object ← ε

ID	*id*

ps-object ← **object**

CONSTRAINTS	*constraints-slot*

slot ← **object**

IDENTIFIED	*set(filler(ps-object))*

(a) single-slot(σ) ← **slot**

CONSTRAINTS	*constraints-slot*
IDENTIFIED	*set(filler(σ))*
ADOPTED	*filler(σ)*

(b) filler(σ) ← **object**

EVALUATIONS	*evaluations-slot*
VALUE	σ

multiple-slot ← **slot**

IDENTIFIED	*set(filler(ps-object))*
ADOPTED	*set(filler(ps-object))*

constraints-slot ← **multiple-slot**

IDENTIFIED	*set(filler(constraint))*
ADOPTED	*set(filler(constraint))*

evaluations-slot ← **multiple-slot**

CONSTRAINTS	*constraints-slot*
IDENTIFIED	*set(filler(evaluation))*
ADOPTED	*set(filler(evaluation))*

objectives-slot ← **multiple-slot**

CONSTRAINTS	*constraints-slot*
IDENTIFIED	*set(filler(objective))*
ADOPTED	*set(filler(objective))*
SELECTED	*set(filler(objective))*
RELEASED	*set(filler(objective))*

Figure 3.2: Upper-level Definitions

of possible recipes by placing constraints on what they are willing to consider. These kinds of meta-decisions can constitute a large chunk of collaborative communication, but most models of dialogue do not represent them explicitly.

To be able to model these and other kinds of decisions-making processes, we add two levels of indirection at each decision point in the model. The first is what we call a *slot*, which contains information about the possible filler values (e.g., recipes) which have been/are under consideration in that context. A slot also contains information about possible constraints which have been put on what should be considered (e.g., not *all* valid recipes, but just those which take less than 30 minutes to execute). A slot also records which (if any) *filler* has been chosen by the agents.

A *filler* is the second layer of indirection. It is used to wrap an actual value with a set of evaluations the agents have made/might make about it. Note that this wrapping is necessary, as evaluations will always be context-dependent (i.e., dependent on the current slot) and cannot, therefore, be attached to the value itself, which may be used simultaneously in various places in the problem solving (e.g., a single truck being used in two separate recipes).

Using these two levels of indirection gives us a rich model not only of the decisions (to be) made, but also of the decision-making process itself.

As the parent of all slots, we define an abstract *slot* type, which is the parent of *single-slot* and *multiple-slot*. These types differentiate decision points where just one filler is needed (e.g., a single recipe for an objective), or where a set of values can be chosen (e.g., objectives that the agents wish to pursue). We first discuss the single-value case, and then the multiple-value case.

Slots for Single Values The most typical case is where a single value can be used to fill a slot. *single-slot* is an abstract type for handling this.[4] It has three features (besides the ID inherited from *object*): IDENTIFIED is the set of all values (wrapped in *fillers*) that the agents have considered/are considering to fill this slot. ADOPTED records the single value which the agents have committed to for this slot. (This value may also be empty in the case that the agents have not yet made a decision, or have reversed a previous decision.) The CONSTRAINTS feature describes possible constraints the agents have put on possible slot fillers (such that the chosen recipe have 5 or fewer steps). Note that this is itself a type of slot, *constraints-slot*, which we will describe shortly.

Note also that the types of the IDENTIFIED and ADOPTED features contain a *filler* type. A *filler* contains both a VALUE which it wraps, as well an EVALUATIONS attribute, which represents any evaluations the agents may make/have made about the value in the local context. This is also a slot of type *evaluations-slot* which we describe below.

Slots for Sets of Values The type *multiple-slot* defines an abstract slot for decisions which allow more than one simultaneous value. In our model, we use three classes which inherit from *multiple-slot*: *constraints-slot*, *evaluations-slot*, and *objectives-slot*. We describe each in turn.

Previously-discussed types *ps-object* and *single-slot* have already introduced the *constraints-slot* type. As discussed above, this type allows a set of constraints to be identified and adopted in a context. Here the IDENTIFIED and ADOPTED features have a similar meaning to those in *single-slot*, with the only exception being that ADOPTED takes a set of *fillers*, instead of a single value.

[4]Note that this is actually a type schema, where σ can be instantiated with a given type. The same holds for the discussion of the *filler* type below.

objective ← ps-object

RECIPE	*single-slot(recipe)*

recipe ← ps-object

ACTIONS	*objectives-slot*
ACTION-CONSTRAINTS	*constraints-slot*

resource ← ps-object

ACTUAL-OBJECT	*id*

constraint ← ps-object

EXPRESSION	*boolean-expression*

evaluation ← ps-object

ASSESSMENT	*unstructured*

c-situation ← ps-object

PENDING-PS-OBJECTS	*set(ps-object)*
PS-OBJECTS	*set(ps-object)*
PS-HISTORY	*list(interaction-act)*
FOCUS	*stack(object)*
OBJECTIVES	*objectives-slot*

Figure 3.3: PS Object Definitions

The second slot type for multiple values is the *evaluations-slot*, which was used above in the definition of *filler*. An *evaluation-slot* provides a space for determining a set of *evaluations*. Its attributes are used in the same way to those of *single-slot* and *constraints-slot* and do not merit further comment here.

The final slot type for multiple values is the *objectives-slot*. It too has the features CONSTRAINTS, IDENTIFIED and ADOPTED which are used as they are in *evaluations-slot*. Objectives are not only committed to, but can also be executed. Objectives in the SELECTED set are those which the agents are currently executing (more details below), as opposed to just intending to execute. Finally, as discussed above, agents must monitor the situation in order to notice when an objective has been fulfilled (so that they stop pursuing it). Objectives which the agents believe have been fulfilled are put into the RELEASED set.

Objective Now that we have described the various slot and filler types which are used in the model, we are ready to get on with the definitions of the abstract PS object types (the formal definitions are shown in Figure 3.3). The type *objectives*, like all six abstract PS objects, inherits directly from *ps-object*. *objective* extends this by adding a RECIPE attribute which is of type *single-slot(recipe)*. This slot provides a place to track all problem-solving activity related to choosing a single *recipe* to use to pursue the *objective*, as

discussed above.

Recipe Recipes are represented as a set of subobjectives (i.e., actions) and a set of constraints on those subobjectives. The ACTIONS attribute is an *objectives-slot* which allows a set of *objectives* associated with the *recipe*, as discussed above. The attribute ACTION-CONSTRAINTS contains the *constraints* placed on the *objectives*.

Constraint *Constraints* are represented as *boolean-expressions*. We do not define the form of these expressions here, but we envision a typical kind of expression involving boolean connectives (*and*, *or*, etc.) as well as (possibly domain-specific) predicates.

Resource *Resources* are used to represent what would typically be thought of as "objects" in a domain. These include real-world objects, but can also include any sort of object used in problem solving that does not fall into one of the other categories of abstract PS objects.

In addition to the attributes inherited from *ps-object*, *resources* contain the attribute ACTUAL-OBJECT, which holds a pointer to the "actual" object as represented in an agent's mental state.

Evaluation Before making decisions in problem solving, agents often evaluate each of the options that have been identified. An *evaluation* represents the agents' assessment of a particular PS object within a particular context. The *evaluation* is therefore always associated with a PS object and a context (e.g., which PS object to choose to fill a slot). As we are not yet sure how best to represent the evaluations themselves, we leave the type of the ASSESSMENT attribute unstructured.

Situation A *c-situation*,[5] describes the state of a possible world, or more precisely, the agents' beliefs about that possible world. Rather than just packing all state information into a general world-state attribute, we separate out information about problem-solving in the situation, and then have a separate place to store other world beliefs.

The PS-OBJECTS attribute holds a set of all PS objects known to the agents in the situation, whereas PENDING-PS-OBJECTS is a temporary holder for objects which have not yet been successfully identified (but have been mentioned in the conversation). These sets include domain-specific PS objects (such as objectives, recipes and resources) which the agents can use in problem solving.

The PS-HISTORY attribute records the history of the agents' problem solving. This is a list of interaction acts the agents have performed (see below).

The OBJECTIVES attribute holds the *objectives* they agents are considering/planning/executing in the situation.

The FOCUS attribute tracks the agents' problem-solving focus as a stack, which is similar to linguistic focus in [GS86].[6]

All other beliefs about the world are stored in the CONSTRAINTS attribute (inherited from *ps-object*). This models the fact that the agents do not have perfect knowledge of the world state, but rather only bits of

[5]This object as well as the CPS acts below are prefaced with *c-* to differentiate them with types in our model of single-agent problem solving, which is not discussed here.

[6]This focus stack has not yet been implemented in the SAMMIE system.

knowledge which constrain which state it is actually in.[7]

Domain Specialization

The CPS model can be specialized to a domain by creating new types that inherit from the abstract PS objects and/or creating instantiations of them. We describe each of these cases separately.

Specialization through Inheritance As described above, inheritance is basically the process of adding new attributes to a previously existing type, and/or specializing the types of preexisting attributes. In our CPS model, inheritance is only used for *objectives*, and *resources*. The other abstract PS objects are specialized through instantiation.

Inheriting from *resource* is done to specify domain-specific resource type. As an example, we define a *ship-by-train* objective for a logistics domain (used in an example below), which is also shown in Figure 3.9 below. This inherits all attributes from *objective* and adds two more: the item to be shipped and a destination which are *single-slots* of type *movable* and *city* respectively (definitions not shown). Similarly, for example, *play-song* in the MP3 domain inherits all attributes from *objective* and adds a has-song slot, which is *single-slot* of type *song*.

Specialization through Instantiation All PS object types (including new types created by inheritance) can be further specialized by instantiation, i.e., by assigning values to some set of their attributes. This can be done both at design time (by the domain modeler) and (as we discuss below) it happens at runtime as part of the problem-solving process itself.

3.2.5 The Collaborative Problem Solving State

The CPS state is part of the agents' common ground [Cla96], and models the agents' current problem-solving context. It is represented as an instance of type *c-situation* called the *actual-situation*. As the name implies, the *actual-situation* is a model of the agents' beliefs about the current situation and the actual problem-solving context.

The OBJECTIVES attribute contains all of the top-level *objectives* associated with the agents' problem solving process. These *objectives* form the roots of individual problem-solving contexts associated with reasoning with, and/or trying to accomplish those *objectives*, and can include all types of other PS objects.

In the overall IS (as shown in Figure 3.1), the CPS state is part of the task information partition of the IS.

Planning and Execution Status

Now that we have described the CPS state and related PS objects, we can, in the next few sections, describe how this allows us to represent the kinds of task information mentioned in Section 3.1.

At the top level, the inclusion of the OBJECTIVES slot in the CPS state allows us to track the planning and execution status of top-level objectives. As noted above, this slot takes a type *objectives-slot* which allows the separation of top-level objectives into the set IDENTIFIED, ADOPTED, SELECTED, and RELEASED.

[7] For reasons of efficiency, we use a back-end database to hold knowledge about resources in the domain within the implemented system.

song ← resource

ID	*id*
CONSTRAINTS	*constraints-slot*
ACTUAL-OBJECT	*id*
TITLE	*single-slot(string)*
ARTIST	*single-slot(artist)*
ALBUM	*single-slot(album)*

Figure 3.4: Definition of *song* within the MP3 Domain

Thus, we are able to distinguish which objectives have been mentioned, which have been committed to, which are currently under execution, and which have been successfully executed.

The definition of *objectives* and *recipes* is recursive (i.e., an *objective* can take a *recipe* which can contain a set of sub-*objectives*, and so forth). This not only allows us to represent hierarchical plans, but also the planning and execution status of objectives at each level. We do this in the same way we keep track of the status of top-level objectives. Similar to the CPS state, a *recipe* also includes an *objectives-slot* in which we can track which possible actions have been identified for the recipe, which have been adopted (i.e., planned), which are currently under execution, and which have successfully been completed.

State of the Decision-making Process

As discussed above, the inclusion of *slots* and *fillers* allows us to track the state of collaborative decision making within a dialogue. Each *slot* contains an IDENTIFIED set which allows it to record all possible values which have been discussed as possible fillers for the slot. This allows us to track several possible alternatives. In addition, each *slot*[8] contains a CONSTRAINTS slot, allowing us to track the constraining of possible values for a slot (e.g., only cities with a beach or a song by The Doors in the earlier examples).

Contextual evaluation of possible values is recorded using the *filler* object, which wraps a possible value with an *evaluations-slot*. Rejection of possible values is handled in update rules which are discussed below in Section 3.3.

Grounding Status of Objects

A more subtle part of the structure allows us to record the grounding status of domain resources in the dialogue. Or, perhaps put in a better way, it allows us to monitor what information about resources is part of the common ground.

To better illustrate this, we show in Figure 3.4 the (simplified) definition of a song within our MP3 domain. Note that here we include features (the first three) inherited from *resource*. Here the *song* type also adds three more slots: TITLE, ARTIST, and ALBUM, which are used to describe that information about the song. (Note that each of these resources also have definitions, which we do not include here.)

An interesting feature of *resources* in general is the ACTUAL-OBJECT slot. As described above, this slot holds a pointer to the system's private knowledge and description of a resource instance. Thus, whereas

[8] Except *constraints-slots* themselves to avoid recursion.

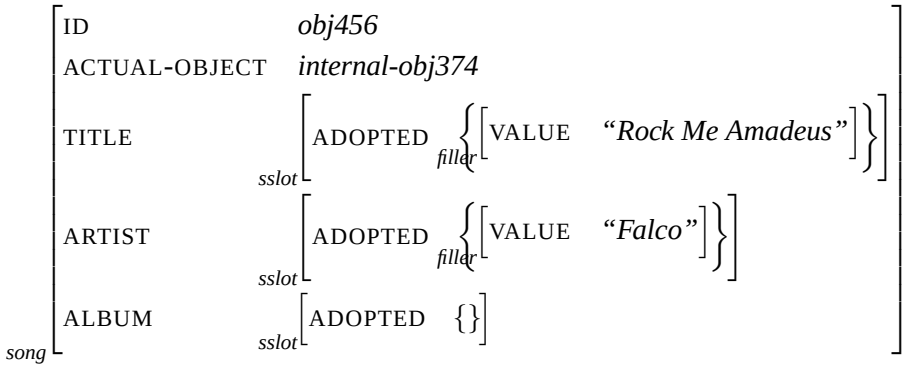

Figure 3.5: *song* instance after Dialogue 13

all other information (except ID) in the resource is meant to describe the common ground, the ACTUAL-OBJECT describes the system's private knowledge about this resource.[9] This essentially allows us to keep track of the *system's* private knowledge about a resource, and the information that is supposedly in common ground.

Consider, for example, the following dialogue:

(13) U: Play the song Rock Me Amadeus from Falco.
S: OK. [starts playback]

After this exchange, the CPS state should have a *song* object that looks something like that shown in Figure 3.5. (Note here we have left out many blank slots of various objects (like IDENTIFIED).) Here we see that the system has grounded this resource into its "world knowledge" as *internal-obj374*. In the implemented system, this internal object contains all information known about the song from the database, including its title, album, artist, length, and file location (so that the MP3 file can be played back). However, with this representation, the system can know that the user does not know (or at least it has not been established as common ground) that, for example, the song being played is on Falco's album "Die Größten Hits". As discussed above, possible uses of this information are the planning of unique referring expressions as well as the generation of additional helpful information.

Task Focus

Attentional focus can be modeled using the FOCUS stack in the CPS state. As noted above, however, we have not yet implemented this in the system.

3.3 Maintaining the Information State

In this final section, we discuss how the IS described above is actually maintained, i.e., how it is updated over the course of a dialogue. We first discuss the maintenance of discourse information, and then the maintenance of task information.

[9] Note that when this slot is not filled, it means that the system has not yet been able to determine or ground in "world knowledge" the resource being discussed. This is used for reference resolution and representing ambiguity in the system, although we do not discuss that here.

3.3.1 Contextual Information

Contextual information is updated by the discourse module Pastis (cf. D5.2) based on input from other modules :

- **last-utterance:** The information of this field is solely based on information from the interpretation module. Individual input from each modality is stored in the *modalities-used* field, the requested modality from user side, if any, is stored in *modality-requested* and the result of interpretation (after multimodal fusion) will be assigned to the *interp* field.
- **discourse-history:** Contextual information that is stored in the discourse history is provided by the interpretation and the output manager, where every contribution of each modality is stored as one *discourse object*.
- **modality-info:** The capabilities of the available modality channels in our setup is loaded when the system is started. In setups where input/output devices can be plugged in and out during runtime this information will be updated dynamically.
- **user-info:** As we have no real-world measurements for a users current cognitive load or a user's expertise in system interaction, the appropriate state values can be set by a graphical control unit. The user/developer currently can select the desired state value for *cognitive-load* and *user-expertise* during runtime. These values are then send via OAA to Pastis in order to update the *user-info*. In a *real world* setup updating the cognitive load state would be based on sensor technology, e.g., tacho sensor, distance sensor in the in-car scenario.

3.3.2 Task Information

Task information is updated by the dialogue manager based on principles of collaborative problem solving. In order to understand this, we must first operationalize the CPS model described above to allow various types of updates. We then describe how this can be used to model the intentions of utterances within a dialogue. We then describe more concretely how the dialogue manager keeps the task information in the IS up-to-date.

3.3.3 Updating the CPS State

Agents change their CPS state through the execution of CPS acts. There are two broad categories of CPS acts: those used in reasoning and those used for commitment. We describe several *families* of CPS act types within those categories:

Reasoning Act Families

- *c-focus*: Used to focus problem solving on a particular *object*.
- *c-defocus*: Removes the focus on a particular *object*.
- *c-identify*: Used to identify a *ps-object* as a possible option in a certain context.

Commitment Act Families

- *c-adopt*: Commits the agents to an *object* in a certain context.
- *c-abandon*: Removes an existing commitment to an *object*.
- *c-select*: Moves an *objective* into active execution.
- *c-defer*: Removes an *objective* from active execution (but does not remove a commitment to it).
- *c-release*: Removes the agents' commitment to an *objective* which they believe has been fulfilled.

Each of these families encompasses a set of CPS acts. For the remainder of this section, we discuss each of the CPS act families and their corresponding acts as well as their effects on the CPS state.

c-focus/c-defocus

Agents need to coordinate their problem-solving focus. They do this through the execution of the following CPS acts (not yet implemented in the SAMMIE system):

c-focus(situation-id,object-id)
c-defocus(situation-id,object-id)

The semantics of these are simple. *c-focus* pushes the given *object-id* onto the focus stack in the *c-situation* represented by *situation-id*. *c-defocus* pops the *object-id* off the stack as well as any *object-ids* above it.

c-identify

CPS acts in the *c-identify* family are used to introduce CPS objects into the realm of a problem-solving context. This could either be in identifying previously unknown objects (i.e., objects not listed in PS-OBJECTS within the *c-situation*), or it could be in identifying a known object as a possible option for filling a certain slot.

All objects must be identified before they can be used further in the CPS process. For this reason, CPS acts in the *c-identify* family exist for all PS objects.[10] The are as follows: *c-identify-objective*, *c-identify-recipe*, *c-identify-constraint*, *c-identify-resource*, and *c-identify-evaluation*.

The basic syntax of identify acts is

c-identify-{*type*}*(slot-id,ps-object)*

where *type* refers to any of the PS objects in the acts listed above. The *ps-object* parameter is the PS object instance which is being introduced and the *slot-id* parameter gives the *id* of the problem-solving context for which it is being identified. Note that *c-identify* acts take an actual *ps-object* as an argument, whereas the remaining CPS acts take only an *object-id* (i.e., pointer to an object).

The effect of a *c-identify* is that the *ps-object* is inserted into the PS-OBJECTS set in the *actual-situation* (if not already there). It is also wrapped in an appropriate *filler* type and inserted into the IDENTIFIED set of the *slot* identified by *slot-id*.

[10] Except *c-situation* for reasons described above.

c-adopt/c-abandon

We treat the CPS act families *c-adopt* and *c-abandon* together here, as one essentially undoes the other. The syntax of the two is as follows:

c-adopt-{type}(slot-id,filler-id)
c-abandon-{type}(slot-id,filler-id)

As with *c-identify*, these two families have types corresponding to most abstract PS objects: *c-adopt-objective*, *c-adopt-recipe*, *c-adopt-resource*, *c-adopt-constraint*, *c-adopt-evaluation*, *c-abandon-objective*, *c-abandon-recipe*, *c-abandon-resource*, *c-abandon-constraint*, and *c-abandon-evaluation*.

A *c-adopt* has the effect of adding the *filler* referred to by *filler-id* to the ADOPTED attribute of the *slot* referred to by *slot-id* (either by assigning the value, in the case of a *single-slot* or adding the value to the set for a *multiple-slot*). Note that this requires that the PS object referred to by *filler-id* be in the IDENTIFIED set in that context.

A *c-abandon* basically has the opposite effect. It removes the object from the ADOPTED attribute. Thus *c-abandon* requires that the object actually be adopted when the act is executed.

When a PS object is adopted with respect to a slot, it means the agents are committed to that object in that context. For example, for a *recipe*, this means the agents are committed to using that *recipe* for the associated *objective*. The other PS objects are similarly treated.

c-select/c-defer

The act families *c-select* and *c-defer* are only used for *objectives*. Their syntax is as follows:

c-select-objective(slot-id,filler-id)
c-defer-objective(slot-id,filler-id)

Executing a *c-select-objective* adds an *objective* to the SELECTED set in the given *objectives-slot*. *c-defer-objective* can then be used to delete an object from the SELECTED set.

Although agents may have any number of adopted *objectives*, there is only a small subset that is actually being executed at any given point. These are the *objectives* in the SELECTED set. An *objective* does not need to be an atomic action to be selected. Higher-level *objectives* can be marked as selected if the agents believe that they are currently executing some action as part of executing the higher-level *objective*.

c-release

The final CPS act we discuss here is *c-release*. As with *c-select* and *c-defer*, this is only applicable to *objectives*. The syntax is as follows:

c-release-objective(slot-id,filler-id)

This act has the effect of moving an *objective* filler from the ADOPTED set to the RELEASED set. Note that the *objective* must first be in the ADOPTED set for this act to be executed.

Rational agents should notice when an *objective* has been successfully achieved and then stop intending to achieve it (cf. [CL90]). The RELEASED set contains those *objectives* which the agents believe have successfully been achieved.

Interaction Acts

An agent cannot single-handedly execute CPS acts to make changes to the CPS state. Doing so requires the cooperation and coordination of both agents. In the model, CPS acts are generated by sets of *interaction acts* (IntActs) — actions that single agents execute in order to negotiate and coordinate changes to the CPS state. An IntAct is a single-agent action which takes a CPS act as an argument.

The IntActs are *begin*, *continue*, *complete* and *reject*. An agent beginning a new CPS act proposal performs a *begin*. For successful generation of the CPS act, the proposal is possibly passed back and forth between the agents, being revised with *continues*, until both agents finally agree on it, which is signified by an agent *not* adding any new information to the proposal but simply accepting it with a *complete*. This generates the proposed CPS act resulting in a change to the CPS state. At any point in this exchange, either agent can perform a *reject*, which causes the proposed CPS act — and thus the proposed change to the CPS state — to fail.

3.3.4 Representing the Intentions of Utterances

In this section, we tie the CPS model described above to language. The CPS model, as presented above can be used as a language for communication between artificial agents, with interaction acts serving as "utterances". There are several things that must be done, however, to make the CPS model usable for modeling *human* communication.

First, we must provide a link from interaction acts to natural language utterances. Second, the CPS model makes the assumption that interaction acts are always successfully received and understood by the other agent. This, of course, is not the case for human dialogue,[11] where mishearing and misunderstanding is more of a rule than an exception. In this section, we provide these necessary parts to turn the CPS model into a dialogue model.

The rest of this section is as follows: first, we deal with the first problem by forming a link between the CPS model and natural language utterances through a definition of *communicative intentions*. We then deal with the second problem by tying the model together with a theory of *grounding*. In order to show the coverage of the model, we show an example of the model applied to a dialogue.

Collaborative Problem Solving and Communicative Intentions

Communicative intentions describe what a speaker wants a hearer to *understand* from an utterance [Gri69]. In this sense, communicative intentions can be very different from a speaker's actual intentions, or what a speaker wants to *accomplish* by making an utterance. This can include things such as manipulation and deception, which are not intended to be perceived by the hearer.

When a speaker has decided on his communicative intentions, he must then encode them in language (e.g., words and sounds) with which they are transfered to the hearer, who then must decode them and (hopefully) recover the original communicative intentions associated with the utterance. No matter what their actual intentions are, speakers will always try to encode their communicative intentions in such a way that they are easily decodable by hearers. This is why communication works, even when one party wants to deceive the other.

[11] It also really is not realistically the case for artificial agent communication either.

In our agent-based dialogue mode, we represent communicative intentions with interaction acts.[12] In other words, each utterance is associated with a set of IntActs. In this way, we are basically modeling dialogue as negotiation about changes to a collaborative problem-solving state. Each utterance is a move in this negotiation, as described with IntActs above.

As an example, consider the following utterance with its corresponding interpretation (in a typical context):

1 A: Let's listen to a song.
begin$_1$*(c-identify-objective(*STATE|OBJVS|ID, 1 *[blank listen-song]))*
begin$_2$*(c-adopt-objective(*STATE|OBJVS|ID, 1 |ID*))*
begin$_3$*(c-focus(*STATE|ID, 1 |ID*))*

where 1 abbreviates an empty *listen-song* objective by using paths into the feature structure.

By assigning these IntActs, we claim that, in the right context, utterance (1) has three communicative intentions (corresponding to the three IntActs):

1. To propose that listening to a song be considered as a possible top-level objective.
2. To propose that this objective be adopted as a top-level objective.
3. To propose that problem-solving activity be focused on the listen-song objective (e.g., in order to specify a song, to find a recipe to accomplish it, . . .).

That these are present can be demonstrated by showing possible responses to (1) which reject some or all of the proposed CPS acts. Consider the following possible responses to (1), tagged with corresponding communicative intentions:

2.1 B: OK.
complete$_1$
complete$_2$
complete$_3$

This is a prototypical response, which completes all three acts.

2.2 B: No.
complete$_1$
reject$_2$
reject$_3$

This utterance rejects the last two CPS acts (*c-adopt-objective* and *c-focus*), but actually completes the first CPS act (*c-identify-objective*). This means that B is actually accepting the fact that this is a *possible* objective, even though B rejects *committing* to it. The next possible response shows this contrast:

2.3 B: I don't listen to songs with clients.
reject$_1$
reject$_2$
reject$_3$

[12] This definition will be expanded in the next section.

Here all three CPS acts are rejected. The *c-identify-objective* is rejected by claiming that the proposed class of objectives is situationally impossible, or inappropriate. Note that it is usually quite hard to reject acts from the *c-identify* family.[13]

2.4 B: OK, but let's talk about where to eat first.
complete$_1$
complete$_2$
reject$_3$

This example helps show the existence of the *identify(c-focus)* act. Here B completes the first two CPS acts, accepting the objective as a possibility and also committing itself to it. However, here the focus move is rejected, and a different focus is proposed (IntAct not shown).

This method of finding responses to reject certain CPS acts proves to be a useful way of helping annotate utterances with their communicative intentions, and we have used it in annotating the example shown in this section.

Grounding

The account of communicative intentions given in the last section is not quite correct. It makes the simplifying assumption that utterances are always correctly heard by the hearer and that he also correctly interprets them (i.e., properly recovers the communicative intentions). In human communication, mishearing and misunderstanding can be the rule, rather than the exception. Because of this, both speaker and hearer need to *collaboratively* determine the meaning of an utterance (i.e., the communicative intentions). This occurs through a process called *grounding* [Cla96].

In this section, we expand our definition of communicative intentions to handle grounding. To do this, we merge our CPS model with a theory of utterance meaning based on Clark's work called *Conversation Acts Theory*. We first introduce Conversation Acts Theory and then use it to expand our definition of communicative intentions.

Conversation Acts Theory

Traum and Hinkelman [TH92] proposed Conversation Acts as an extension to speech act theory. Similar to our provisional model of communicative acts, theories based on speech acts typically made the assumption that utterances are always heard and understood. To overcome this, Traum and Hinkelman defined Conversation Acts on several levels, which describe different levels of the communicative process. Table 3.1 shows the different levels and acts, which we briefly describe here and then show an example using Conversation Acts.

Core Speech Acts A major contribution of Conversation Acts is that it takes traditional speech acts and changes them into Core Speech Acts, which are multiagent actions requiring efforts from the speaker

[13] In our current CPS model, when a *c-identify* is rejected, the corresponding object then only exists in the PS-HISTORY attribute of the *actual-situation*. It is likely that it would be beneficial to record this somewhere within the corresponding slot as well (e.g., to make sure the same suggestion isn't made twice). We leave this question to future research.

Discourse Level	Act Type	Sample Acts
Sub UU	**Turn-taking**	take-turn keep-turn release-turn assign-turn
UU	**Grounding**	Initiate Continue Ack Repair ReqRepair ReqAck Cancel
DU	**Core Speech Acts**	Inform WHQ YNQ Accept Request Reject Suggest Eval ReqPerm Offer Promise
Multiple DUs	**Argumentation**	Elaborate Summarize Clarify Q&A Convince Find-Plan

Table 3.1: Conversation Act Types [TH92]

and hearer to succeed. To do this, they define a *Discourse Unit* (DU) to be the utterances which contribute to the grounding of a Core Speech Act. These are similar to Clark's *contributions* [Cla96].

Argumentation Acts Conversation Acts also provides a place for higher-level *Argumentation Acts* which span multiple DUs. As far as we are aware, this level was never well defined. Traum and Hinkelman give examples of possible Argumentation Acts, including such things as rhetorical relations (e.g., [MT87]) and plan construction plans (e.g., [LA90]).

Grounding Acts The most important part of Conversation Acts for our purposes here are *Grounding Acts* (GAs). These are single-agent actions at the *Utterance Unit* (UU) level, used for the grounding process. The GAs are as follows:

Initiate The initial part of a DU.

Continue Used when the initiating agent has a turn of several utterances. An utterance which further expands the meaning of the DU.

Acknowledge Signals understanding of the DU (although not necessarily *agreement*, which is at the Core Speech Act level).

Repair Changes some part of the DU.

ReqRepair A request that the other agent repair the DU.

ReqAck An explicit request for an acknowledgment by the other agent.

Cancel Declares the DU as 'dead' and ungrounded.

These form part of Traum's computational theory of grounding [Tra94], which uses finite state automata to track the state of grounding for DUs in a dialogue.

$GA_{DU\#}$	UU#	: Utterance
$init_1$	1.1	U: okay, the problem is we better ship a boxcar of oranges to Bath by 8 AM.
ack_1	2.1	S: okay.
$init_2$	3.1	U: now ... umm ... so we need to get a boxcar to Corning, where
		: there are oranges.
$init_3$	3.2	: there are oranges at Corning
$reqack_3$	3.3	: right?
ack_3 $init_4$	4.1	S: right.
ack_4 $init_5$	5.1	U: so we need an engine to move the boxcar
$reqack_5$	5.2	U: right?
ack_5 $init_6$	6.1	S: right.

Figure 3.6: Example of Conversation Acts: Grounding Acts [TH92]

DU#	Core Speech Act types	Included UUs
1	$inform^U$ $suggest(goal)^U$ $accept^S$	1.1 1.2
2	$inform^U$ $suggest^U$	3.1
3	$check^U$ $?suggest^U$	3.2 3.3 4.1
4	$inform\text{-}if^S$ $?accept^S$	4.1 5.1
5	$check^U$	5.1 5.2 6.1

Figure 3.7: Example of Conversation Acts: Core Speech Acts [TH92]

Turn-taking Acts At the lowest level are *Turn-taking Acts*. These are concerned with the coordination of speaking turns in a dialogue. A UU can possibly be composed of several Turn-taking Acts (for example, to take the turn at the start, hold the turn while speaking, and then release it when finished).

Example

As an example, Traum and Hinkelman annotate part of a dialogue from the TRAINS-91 corpus [Gat92] to illustrate Conversation Acts. In Figures 3.6 and 3.7, we show a section of their example of GA and Core Speech Act annotations.[14]

Figure 3.6 shows the GAs associated with each UU (subscripted with the number of the DU they contribute to). Figure 3.7 shows the Core Speech Acts performed in each DU (superscripted with the initiating party).

Discussion One main contribution of Conversation Acts Theory is that it models dialogue with utterances making simultaneous contributions at several different levels. While we believe that dialogue should be modeled as several levels, we see several difficulties with the theory as it now stands.

First, the theory only specifies act *types* at the various levels, but not their content. This is true even at the interface between levels. For example, GAs are modeled as negotiation about the meaning of a DU, but it is unclear exactly which part the meaning of DU 1 (if the $init_1$ in the example is initializing (*inform*,

[14]The original used S for the system and M for 'manager'. We have changed this to the more typical U for 'user'.

suggest, *accept*)).

Also, although this model improves on speech act theory by modeling speech acts as multiagent actions, it still suffers from some of the difficulty of speech acts. In particular, Conversation Acts Theory does not attempt to define a (closed) set of allowable Core Speech Acts. It is in fact unclear if such a closed set of domain-independent speech acts exists (cf. [Cla96, DJTM97]). In practice, this fact has lead to many different proposed taxonomies of speech or dialogue acts, many of which are domain dependent (e.g., [AC97, ABWF$^+$98] — also cf. [Tra00]).

Finally, as mentioned above, the Argumentation Acts level was never well defined. In [TH92], Argumentation Acts are described vaguely at a level higher than Core Speech Acts which can be anything from rhetorical relations to operations to change a joint plan.

Defining the Dialogue Model

In constructing our agent-based dialogue model, we first take Conversation Acts as a base. In particular, we model communicative intentions at several simultaneous levels, which more or less correspond to those used in Conversation Acts. In addition to this, we expand and concretize several of the levels using the collaborative problem-solving model discussed above. This allows us to overcome several of the difficulties of Conversation Acts mentioned above.

The levels we model are: Turn-taking Acts, Grounding Acts, Interaction Act, and CPS Acts. We discuss each in turn and then reinterpret the previous example with our model.

Turn-taking Acts At the sub-utterance level, we use the turn-taking model as it is in Conversation Acts. We will not refer to it further in our examples, as it is not the focus of our work.

Grounding Acts At the utterance unit level (UU), we use the Grounding Act *types* as they are defined in Conversation Acts. We extend this and define *contents* for these acts, namely an Interaction Act.[15]

Interaction Acts At the discourse unit level (DU), we depart from Conversation Acts. Instead of using Core Speech Acts, we use IntActs, as described in Section 3.3.4. Unlike Core Speech Acts, these are not just labels, but also contain content (instantiated CPS acts).

CPS Acts Finally, we propose the use of CPS acts at the level of Argumentation Acts. These are a natural fit, as a number of IntActs need to be executed by different agents in order to generate a CPS act, which then makes changes to the CPS state. This gives us a natural segmentation of discourse units.

TRAINS Example Revisited

To show concretely, how these (last three) levels fit together, we revisit Traum and Hinkelman's example from above, interpreting it in the agent-based model. Figure 3.8 shows the dialogue marked up with

[15] A Grounding Act could also theoretically take another Grounding Act as an argument, as in meta-repairs and so forth [Tra94]. For simplicity, we have decided to avoid these cases at this stage. We hope to add support for meta-grounding in the future.

instantiated grounding acts. We first discuss the dialogue at the grounding level, and then at the problem-solving level.

Grounding Our analysis at the grounding level is basically unchanged from that of Traum and Hinkelman as shown in Figure 3.6. We therefore only briefly describe it. The main difference between the two accounts is that we associate a GA with each individual IntAct, and therefore have more instances in several cases.

In UU 1.1, the user initiates three IntActs, which the system acknowledges in UU 2.1. Note that, only at this point are the effects of the IntActs valid (see discussion above). This means that, only after UU 2.1 are, for example, IntActs 1, 2 and 3 placed in the PS-HISTORY list of the CPS state.

UU 2.1 also initiates the corresponding *complete* IntActs to those initiated in UU 1.1; these are acknowledged in UU 3.1.

UU 3.1 also inits three IntActs (7, 8, and 9), which are never grounded,[16] and thus do not result in any successfully executed IntActs, and thus no changes to the CPS state.

In UU 3.2, the user inits new IntActs for which he explicitly requests an acknowledgment in UU 3.3. Finally, UU 4.1 inits two completes, which are acknowledged by the subsequent utterance by the user (not shown). Note that, again, these completes are not valid until grounded. Thus the CPS state after UU 4.1 will reflect the state as if those completes did not yet occur.

Problem Solving At the problem-solving level, the user proposes the adoption (and identification) of an objective of shipping oranges in UU 1.1. (Again, these do not become active until grounded in UU 2.1). He also proposes that problem-solving focus be placed on that objective (i.e., in order to work on accomplishing it). The proposed objective is shown in Figure 3.9, and deserves some explanation.

The type of the objective is *ship-by-train*, which we have just invented for this example.[17] It introduces two new attributes to the *objective* class: an item to be shipped and a destination. As the abbreviated form of the objective shows, there are three main components to the objective as it has been introduced by the user. First of all, it has a pre-adopted destination — Bath (modeled as a location with a NAME). Second, the item to be shipped has not yet been determined, but a constraint has been put on possible values for that slot — they must be of type *oranges*.

Recall from the discussion above that this was one of the motivations for introducing slots in the model. Notice here that the constraint is not put on a particular instance of oranges, rather it is put on the *single-slot* itself. Constraints on a slot are adopted to restrict the values considered (e.g., identified) as possible fillers.

Finally, a constraint has also been placed on the objective itself — that it be completed by 8 a.m. Note the difference here between placing a constraint on a *ps-object* versus placing it on a *slot*, as just discussed. As mentioned in the previous section, objectives and resources are usually extended by the addition of additional resources (as we did here with the item and destination). However, it is also possible to further define an objective or resource by placing a constraint on it. This seems to work well in cases like this from natural language, where, for example, an adverb is used. Of course, it would also possible to add an extra attribute to the type for completion time. This decision must be made by the domain designer.

[16] Although see discussion in [TH92].

[17] In all examples, we invent simple-minded domain-specific object types as we need them.

1.1 U: okay, the problem is we better ship a boxcar of oranges to Bath by 8 AM.
init$_1$*(begin*$_1$*(c-identify-objective(*STATE|OBJVS|ID, [1] *)))*
init$_2$*(begin*$_2$*(c-adopt-objective(*STATE|OBJVS|ID, [1] |ID*)))*
init$_3$*(begin*$_3$*(c-focus(*STATE|ID, [1] |ID*)))*

2.1 S: okay.
ack$_1$
ack$_2$
ack$_3$
init$_4$*(complete*$_1$*)*
init$_5$*(complete*$_2$*)*
init$_6$*(complete*$_3$*)*

3.1 U: now ... umm ... so we need to get a boxcar to Corning, where there are oranges.
ack$_4$
ack$_5$
ack$_6$
init$_7$*(begin*$_4$*(c-identify-recipe(* [1] |REC|ID, [2] *)))*
init$_8$*(begin*$_5$*(c-adopt-recipe(* [1] |REC|ID, [2] |ID*)))*
init$_9$*(begin*$_6$*(c-focus(*STATE|ID, [2] |ID*)))*

3.2 U: there are oranges at Corning
init$_{10}$*(begin*$_7$*(c-identify-constraint(*STATE|CONS|ID, [3] *)))*
init$_{11}$*(begin*$_8$*(c-adopt-constraint(*STATE|CONS|ID, [3] |ID*)))*

3.3 U: right?
reqack$_{10}$
reqack$_{11}$

4.1 S: right.
ack$_{10}$
ack$_{11}$
init$_{12}$*(complete*$_7$*)*
init$_{13}$*(complete*$_8$*)*

Figure 3.8: The TRAINS Example Interpreted with the Agent-based Model

$$\boxed{1}\ \textit{ship-by-train}\left[\begin{array}{lll}\text{CONS}|\text{APTD} & \textit{fill}\left\{\left[\text{VAL}|\text{EXPR} \quad \textit{'completion before 8 a.m.'}\right]\right\} \\ \text{ITEM}|\text{CONS}|\text{APTD} & \textit{fill}\left\{\left[\text{VAL}|\text{EXPR} \quad \textit{'type: oranges'}\right]\right\} \\ \text{DEST}|\text{APTD} & \textit{fill}\left[\text{VAL}|\text{NAME} \quad \textit{'Bath'}\right]\end{array}\right]$$

Figure 3.9: Contents of *objective* [1]

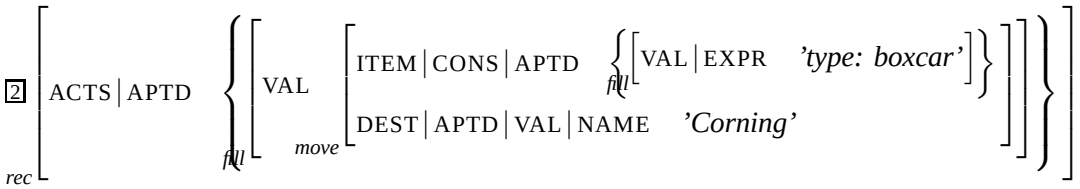

Figure 3.10: Contents of *recipe* 2

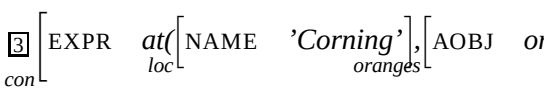

Figure 3.11: Contents of *constraint* 3

UU 2.1 completes the CPS acts, and after 3.1, when the completes are grounded, the CPS acts are generated, resulting in the corresponding changes to the CPS state.

In UU 3.1, the user proposes the adoption of a (partial) recipe for shipping the oranges, as well as that focus be placed on it. These IntActs are never grounded, and thus never result in a change in the CPS state. However, as this utterance gives a good example of the introduction of a recipe, however, we will still discuss it. The recipe that the user attempts to introduce is shown in Figure 3.10.

This recipe consists of a single adopted objective — that of moving a boxcar to Corning. Similar to the *ship-by-train* objective above, this also has an adopted value (the destination is Corning), and a constraint on the slot of the other (that the only resources to be considered for the move should be of type *boxcar*). At this point, the recipe has no ACTION-CONSTRAINTS.

However, this recipe never makes it into the CPS state (even as something mentioned). Instead, the user decides he wants to adopt (confirm) the joint belief that there are oranges at Corning, as reflected in UU 3.2. Note that no focus change is proposed by UU 3.2. We model this in this way, as it appears that the user did not intend for further work (beyond adoption) to be done on this constraint, or on finding out the state of the world. Instead, it was intended as a quick check, but focus was intended to remain on the *ship-by-train* objective.

As discussed above, we model beliefs about the world state as constraints on the situation. The mentioned constraint is shown in Figure 3.11. We do not yet have a theory of constraint representation, thus we have been glossing constraints until now. The only specification we have made is that the EXPRESSION be of type *boolean*. In the case of this constraint, however, a simple gloss is not enough, as this constraint actually introduces a new embedded *resource* — the instance of the oranges that are at Corning. For this reason, we show this constraint as a domain-specific predicate (*at*) that takes a location and an item. It is obvious that work needs to be done on the general specification of constraints, but the representation here is sufficient for our purposes.

It is important to point out that when the *beginc-identify-constraint* is grounded in UU 4.1, the oranges instance from the constraint is also placed in the PS-OBJECTS set within the CPS state, making it available for use in further problem solving.

Discussion Our CPS dialogue model overcomes several of the difficulties with Conversation Acts we mentioned above. First, it defines act types as well as the *content* of those acts. Second, it defines a closed set of domain-independent acts at discourse unit level (i.e., IntActs with CPS acts as arguments). Although the acts are domain-independent, they can be used with domain-specific content through PS

object inheritance and instantiation). Finally, the model introduces CPS acts at the level of Argumentation Acts, which define intentions for larger chunks of discourse.

As promised above, we are now able expand our definition of communicative intention — this time to a fully-instantiated grounding act. As illustrated in the example above, we use these to represent the intended meaning of an utterance — both to do work at the grounding level (e.g., acknowledging previous IntActs) and to introduce new IntActs to be grounded. This is similar to Clark's proposed communication *tracks* [Cla96].

3.3.5 Dialog Manager Update Rules

As described in Deliverable 2.1 [MAB+05], dialogue management in the SAMMIE system can be grouped into three separate processes:

Integrating Utterance Information Here the system integrates grounding acts — by both user and system — as they are executed.

Agent-based Control Once executed grounding acts have been integrated into the dialogue model, the system must decide what to do and what to say next.

Package and Output Communicative Intentions During the first two phases, communicative intentions (i.e., instantiated grounding acts) are generated, which the system wants to execute. This last phase packages these and sends them to the generation subsystem for realization. When realization is successful, the information state is updated using the rules from the first phase.

In the first stage, communicative intentions from the user (in the form of Grounding Acts) are integrated into the IS according to the general CPS State update rules described above. For example, if the Grounding Act is an *ack*, then the embedded Interaction Act is "executed" and its effects result in IS update. If that Interaction Act is a *complete*, then the embedded CPS Act is "executed" and its effects are applied to the state. If the CPS Act is a *c-identify*, then this means the corresponding PS Object is added to the IDENTIFIED set of the slot pointed to by the context parameter of the CPS Act, and so forth.

In the last stage, a similar process happens to integrate communicative intentions from the system. This actually happens *after* the system has successfully realized the output. When the dialogue manager sends off a set of Grounding Acts to the generation subsystem, it records them in the PENDING-SYS-UTT list in the information state. When successful output has been completed, the Output Manager notifies the dialogue manager, and then the dialogue manager removes these Grounding Acts from the list and updates the IS given the system's turn.

3.4 Summary

In this chapter we described the extended information state used in the SAMMIE system. We first discussed what extensions with respect to typical information state are needed, given our research interests in WP3 and WP2, namely contextually adaptive and flexible multimodal presentation and generic task modeling in the collaborative problem solving paradigm. Accordingly, the extensions concern the representation of contextual information pertaining to multimodal discourse and task representation as the collaborative problem solving state. The contextual information involves, in particular, a record of the latest utterance

and preceding discourse history (containing representations of discourse entities introduced in the different modalities), information about available modalities and their capacity to convey certain amounts of information, and information about the user's cognitive load, expertise and modality preferences. We use the contextual information both for context-sensitive interpretation of user utterances, and for contextually adaptive multimodal presentation (cf. deliverable D3.2). The collaborative problem solving state tracks problem solving objects, such as objectives, recipes, resources and constraints on these, as the dialogue progresses. It provides a record of task planning and execution status, the decision-making process and grounding status of objects. After discussing the information needed in general, we described how this information is represented in a structured way in the extended information state in the SAMMIE system and how these representations are maintained by update rules in the course of a dialogue. The prototype included in the appendix contains the current implementation of the extended information state within a preliminary version of the SAMMIE final showcase.

Chapter 4

The GoDiS Extended Information State

This chapter describes the modelling of the extended information state for the appropriate generation of multimodal contributions in the GoDiS system, based on theories of information structure. The GoDiS information state developed for unimodal contexts and used in a number of applications, is here extended with various material that enables the representation of available modalities, the determination of information structure, and other factors that are to be used in the subsequent determination of the appropriate distribution of a system contribution across different modalities.

The application in question is a public transport or tram information system, which can be used both in the home and in the car, as will be described below. The chapter begins with a theoretical discussion of the information that is needed in the GoDiS information state for a multimodal tram system, in section 4.1. Section 4.2 then discusses how this information is to be represented in the information state, and section 4.3 describes the update rules needed to maintain the extended information state.

4.1 Information in the extended information state

This section first of all gives a view of the GoDiS information state in applications prior to the consideration of multimodal generation. Next, interface variables and update rules in GoDiS are described. The section then describes the type of information needed in the GoDiS information state for multimodal generation in a tram application, and specifically the information that is needed in addition to information already in place in the information state.

4.1.1 A basic GoDiS information state

Figure 4.1 shows the GoDiS information state (IS). There is a basic division of IS into a record of information private to the system, and a record representing shared information.

Agenda The field /PRIVATE/AGENDA is of type Stack(Action). In general, we try to use data structures which are as simple as possible; a stack is the simplest ordered structure so it is used as a default data structure where order is needed as long as it is sufficient for the purposes at hand. The agenda is read by the selection rules to determine the next dialogue move to be performed by the system.

$$
\left[\begin{array}{lll}
\text{PRIVATE} & : & \left[\begin{array}{lll}
\text{AGENDA} & : & \text{OpenQueue(Action)} \\
\text{PLAN} & : & \text{OpenStack(PlanConstruct)} \\
\text{BEL} & : & \text{Set(Prop)} \\
\text{TMP} & : & \left[\begin{array}{lll} \text{USR} & : & Tmp \\ \text{SYS} & : & Tmp \end{array}\right] \\
\text{NIM} & : & \text{OpenQueue(Pair(DP, Move))}
\end{array}\right] \\
\text{SHARED} & : & \left[\begin{array}{lll}
\text{COM} & : & \text{Set(Prop)} \\
\text{ISSUES} & : & \text{OpenStack(Question)} \\
\text{ACTIONS} & : & \text{OpenStack(Action)} \\
\text{QUD} & : & \text{OpenStack(Question)} \\
\text{PM} & : & \text{OpenQueue(Move)} \\
\text{LU} & : & \left[\begin{array}{lll} \text{SPEAKER} & : & \text{Participant} \\ \text{MOVES} & : & \text{OpenQueue(Move)} \end{array}\right]
\end{array}\right]
\end{array}\right]
$$

$$
Tmp = \left[\begin{array}{lll}
\text{COM} & : & \text{Set(Prop)} \\
\text{ISSUES} & : & \text{OpenStack(Question)} \\
\text{ACTIONS} & : & \text{OpenStack(Action)} \\
\text{QUD} & : & \text{OpenStack(Question)} \\
\text{AGENDA} & : & \text{OpenQueue(Action)} \\
\text{PLAN} & : & \text{OpenStack(PlanConstruct)}
\end{array}\right]
$$

Figure 4.1: Information State type in GoDiS for action-oriented dialogue

Plan The /PRIVATE/PLAN is a stack of plan constructs. Some of the update rules for managing the plan have the form of rewrite rules which process complex plan constructs until some action is topmost on the plan. Other rules execute this action in case it is a system action or move it to the agenda in case it is a move-related action.

Private beliefs In GoDiS, the field /PRIVATE/BEL, a set of propositions, is used to store the results of database searches. Of course, the database (and the domain knowledge, and the lexicon) can be seen as a part of the system's private belief set, but in /PRIVATE/BEL we choose to represent only propositions which are directly relevant to the task at hand and which are the result of database searches. This is similar to seeing the database as a set of implicit beliefs, and database consultation as an inference process where implicit beliefs are made explicit. The reason for using a set is that a set is the simplest ordered data structure.

Questions Under Discussion In GoDiS we define Questions Under Discussion, or QUD, to be an open stack of questions that can be addressed using short answers. The open stack has some set-like properties, but also retains a stack structure in case it should be useful for ellipsis resolution[1].

[1]Note that this is different from the way Ginzburg [Gin96b] defines QUD, i.e. as containing questions which have been raised but not yet resolved, and thus currently under discussion. For reasons given in chapter 5 of [Lar02], we have divided Ginzburg's QUD into two structures: QUD and Issues.

Issues The field ISSUES contains all questions which have been raised in a dialogue (explicitly or implicitly) but not yet resolved. It thus contains a collection of current, or "live" issues. A suitable data structure appears to be an open stack, i.e., a stack where non-topmost elements can be accessed. This allows a non-rigid modelling of current issues and task-related dialogue structure.

Actions The actions stack is an open stack, which is the same structure that we use for ISSUES. It contains actions which have been requested but not yet completed.

Shared Commitments The field /SHARED/COM contains the set of propositions that the user and the system have mutually agreed to during the dialogue. They need not actually be believed by either participant; the important thing is that the DPs have committed to these propositions, even if only for the purposes of the conversation.

To reflect that the contents need not be true, or even privately believed by the DPs, and because we are not using situation semantics (where there is a distinction between facts and propositions) we use the label "commitments" or "committed propositions", abbreviated as COM, instead of FACTS. These, then, are propositions to which the DPs are (taken to be) jointly committed.

Latest utterance In /SHARED/LU we represent information about the latest utterance: the speaker, and the moves realized by the utterance.

Temporary store To enable the system to backtrack if an optimistic assumption turns out to be mistaken, relevant parts of the information state is kept in /PRIVATE/TMP. The QUD and COM fields may change when integrating an ask or answer move, respectively. The plan may also be modified, e.g., if a raise action is selected. Finally, if any actions are on the agenda when selection starts (which means they were put there during by the update module), these may have been removed during the move selection process.

Non-integrated moves Since several moves can be performed per turn, GODIS needs some way of keeping track of which moves have been interpreted. This is done by putting all moves in LATEST_MOVES in a queue structure called NIM, for Non-Integrated Moves. This structure is private, since it is an internal matter for the system how many moves have been integrated so far. Once a move is assumed to be grounded on the understanding level the move is added to the /SHARED/LU/MOVES set. Since the move has now been understood on the pragmatic level, the content of the move will be a question or a full proposition (for short answers, the proposition resulting from combining it with a question on QUD).

Previous moves To be able to detect irrelevant followups, GODIS needs to know what moves were performed (and grounded) in the previous utterance. These are stored in the /SHARED/PM field.

4.1.2 Interface variables and update rules

In addition to what has been described so far, the GODIS Total Information State (TIS) also contains resource interface variables (RIVs) and module interface variables (MIVs). The RIVs provide interfaces to lexicon, domain, and device resources, and the like, so that these can be accessed from the modules

through the (total) information state. The MIVs similarly provide interfaces for different variables that the modules read from and write to. An example of a basic set of MIVs for GODIS is given in figure 4.2.

INPUT : String
OUTPUT : String
LATEST_SPEAKER : Participant
LATEST_MOVES : Oqueue(Dmove)
NEXT_MOVES : Oqueue(Dmove)
PROGRAM_STATE : Program_state
SCORE : Real
TIMEOUT : Real
LANGUAGE : Language

Figure 4.2: Module interface variables in GODIS

Figure 4.2 first of all shows (user) INPUT and (system) OUTPUT in the form of strings, corresponding to the latest user and system utterances, respectively. The participant as the value of LATEST_SPEAKER is either the user or the system, as appropriate at any given point during the dialogue. The LATEST_MOVES is an open queue of dialogue moves corresponding to the latest turn in the dialogue, either by the user or the system, and the NEXT_MOVES contains the moves that the system is going to make next. The PROGRAM_STATE records whether the system is running or is about to stop. Then there are two variables taking values of type Real. The first of these is the recognition SCORE, which is either given by a speech recogniser, or can be simulated through the user typing in a value, if a text mode is used for input. The second is a variable TIMEOUT that enables the specification of an amount of time after which the system is to take the initiative if the user has not said anything. The final module interface variable gives the LANGUAGE that is currently being used in the dialogue.

The total GODIS information state that has now been described, needs to be extended in different ways in order to include considerations of multimodality. However, before turning to this issue, we also need to consider *update rules* in GODIS, as these will be extended and modified below. The GODIS Update module makes use of a number of different update rules for updating the information state during a dialogue. The updates are based on the content of user and system utterances, as well as on different actions in the private part of the information state. Every update rule belongs to a rule class. There are, for instance, update rules of the class integrate that are concerned with integrating different moves in the information state, such as user ask moves, system ask moves, and so on. Other examples of rule classes are downdate_qud and downdate_issues that manage the QUD and ISSUES structures, and exec_plan that handles different actions in the PLAN.

Each update rule consists of a number of preconditions, which have to be fulfilled for the application of the rule, and a number of effects, which are the result of the application of the rule. As an example, consider the update rule **integrateGreet** in figure 4.3

The update rule **integrateGreet** contains three preconditions. These involve the NIM being empty, the assignment of the first element in NIM to a variable DPM, and the second element in DPM encoding a greet move. If these preconditions are met, the two effects pop NIM, thereby removing the greet move from the non-integrated moves, and add the greet move among the moves for the latest utterance, thereby

RULE: **integrateGreet**
CLASS: integrate
PRE:
not empty($/private/nim)
$/private/nim/fst = DPM
DPM/snd = greet
EFF:
pop(/private/nim)
add(/shared/lu/moves, DPM/snd)

Figure 4.3: The GoDiS update rule **integrateGreet**

integrating this move. Update rules are discussed in detail by [Lar02].

Let ut now turn to the issue of what information needs to be added to the information state for multimodal generation.

4.1.3 Information needed for multimodal generation

The overarching research question that UGOT addresses in work package three is the determination of in which modalities different parts of a given proto-content are to be realised. In order to facilitate the reader's comprehension of the extensions to the GoDiS information state described below, we begin by some concrete examples illustrating the challenge involved in the overarching research question that we address.

Consider a tram information system as a prototype application for the exploration of these issues. The output modalities of this system can be assumed to be speech and a graphical display in the form of a map.[2] The functionalities of this application include asking what time some tram line leaves or arrives at a particular stop.

The exchange in (14)-(17) below gives an illustration of the issues at hand, where (14) is a user contribution, and (15)-(17) are some of the possible system responses. Example (14) shows a user question, and a simple representation of the proto-content – corresponding to the "full message" – of the system's ensuing response:

(14) *U: What time is the next tram from Marklandsgatan to Brunnsparken?*
Proto-content of system contribution to be generated:
$next\_tram(marklandsg, brunnsp, 14.23)$

The task of the system is then to determine an appropriate contextualised contribution for the proto-content in (14), which includes the determination of in which modalities – here speech and graphics – different parts of the message are to be realised. The examples below illustrate a few different possible contextualised system contributions for the current dialogue exchange.

[2]This graphical display is the OAA map agent developed and described in TALK deliverable D5.1.

Example (15), first of all, illustrates a system response where the whole message is simultaneously realised both through speech and graphically:

(15) Say everything, show everything:

S: The next tram from Marklandsgatan to Brunnsparken is at 2.23pm

< The points on the map corresponding to the tram stops Marklandsgatan and Brunnsparken are highlighted, as well as the path between them. The time "2.23pm" is also written on the graphical display. >

The next example illustrates the spoken and graphical modalities realising different pieces of the message. Modalities realising less than the full message make use of an information structural distinction whereby the *focus* constitutes the informative part of the contribution, and the *ground* corresponds to the backgrounded part, that part which is a reflection of the context. These terms are used here in a way that roughly corresponds to their use by [Val92]. For a discussion of the differences between Vallduví's and our use of the terms, see [Eri05].

Turning now to the actual dialogue example, the alternative system response in (16) involves realising the focus through speech, and the whole message (both focus and ground) through graphics:

(16) Say the focus, show everything:

S: 2.23pm

< The points on the map corresponding to the tram stops Marklandsgatan and Brunnsparken are highlighted, as well as the path between them. The time "2.23pm" is also written on the graphical display. >

A third possibility for the system response is to realise the ground in one modality and the focus in the other modality. Example (17) shows the system giving the ground through speech, and the focus graphically:

(17) Say the ground, show the focus:

S: The next tram from Marklandsgatan to Brunnsparken is at

< The time "2.23pm" is written on the graphical display. >

Based on the information structural primitives *ground* and *focus*, it is possible for each modality either to realise the full message (corresponding to both the focus and the ground), just the ground, just the focus, or nothing at all. For a dialogue system application with n number of output modalities there are therefore, at least *a priori*, 4^n possible contextualisations of a given proto-content. For the tram system considered here, there are thus 16 possible contextualisations to consider.[3] The issue discussed in the present chapter is what information needs to be added to the information state in order for other components of the system to be able to determine which possible contextualisation is to be used for a given proto-content and context.

In a general way, the information state first of all needs to record information about what modalities are available for a particular application. For the tram information application, this means that the information state needs to be extended with the information that speech and a graphical display are available. As the UGOT work in work package three focuses on the generation perspective, the considerations in this

[3] One of these possibilities involves nothing being said and nothing being shown graphically. As this is a highly infelicitous contribution, not conveying any part of the message at all, one may want to exclude this possibility from consideration. In practice one may therefore wish to consider only $4^n - 1$ possibilities, here 15.

chapter concern output, and output modalities. However, if both input and output modalities are being considered, and specifically if there may be differences between what modalities are used for input and what modalities are used for output, then the information state needs to distinguish input modalities from output modalities.

One may imagine a few different ways in which the information about available modalities is to be conveyed to the information state. One is that this information is hard-wired in the application by the application developer, in the sense that the developer extends the information state with this information as the application is designed. A more flexible but a great deal more complex way is to allow this information to be changed dynamically, when the system is being used. Ideally this would be done through plug'n'play, so that when a new output device is connected to the system, a representation of this modality is also added to the information state. The implementation below takes the first approach, and information about available modalities is added by the system developer, but this is not to be seen as excluding the second option as something to explore.

In addition to information about available modalities, the information state also needs to be extended with information about the extent to which a given message is to be realised in any one particular modality (for all available modalities). Using the information structural notions introduced above, the basic possibilities here thus involve the following:

(18)
- The full message, corresponding to both focus and ground
- Just the ground
- Just the focus
- Nothing at all

The complete exclusion of an output modality – which may either be seen as the "Nothing at all" option in (18) or as a certain modality's not being available for a particular dialogue or part of a dialogue – may result from constraints placed by the physical context. For the tram system one may for instance imagine a speech-only mode where graphics is not used for system output. This mode would be used, say, in communication with the application over a telephone or a mobile phone, in a car while one is driving (in countries where this is legal), or at home whenever one is unable to look at a screen. One may also imagine a graphics-only mode where no speech is used for system output. This mode can be used in any kind of situation where speech is to be avoided, such as in the home or in the car if one does not want others to hear or be disturbed, or in a meeting or in an audience during a talk or a film screening, and so on.

Such modes can be seen as default behaviour of the system in different kinds of situations, and it is therefore desirable that the system enables the setting of defaults with regard to output behaviour. Such defaults may also concern the other degrees of realisation in (18) above. It may for instance be part of the "personality" of a system that, unless constrained by other factors, it is highly explicit about the focus material, and therefore always gives the focus in all modalities.

In addition to speech-only and graphics-only modes for the tram application (and other applications more generally), one may also envisage different situations in which both modalities are available. One such situation is a user in a stationary car, where she may both be able to hear the system and watch graphical output on a display. Another scenario is the user being at home in front of a computer, or anywhere on the move with a PDA.

Other factors besides physical context and default behaviour such as personality traits, may naturally also influence the degree to which a message is realised in some modality. Of importance to the detailing

of the extended information state as discussed in the current chapter, is also the factor of communication difficulties. Specifically, speech recognition scores for spoken user contributions can be taken into account here. Consider the user question in (14) again, repeated here as (19):

(19) U1: What time is the next tram from Marklandsgatan to Brunnsparken?

If the recognition score for $U1$ is high – the system therefore being fairly certain of having heard the user correctly – the system may realise only the focus, "2.23pm", either using speech, or graphics, or both. However, if the recognition score is lower – below some threshold to be determined empirically – the system may choose also to realise (what it believes to be) the ground, thereby making it explicit to the user what it has understood, and giving the user a chance to correct the system, as in (20):

(20) $<$ recognition score for $U1$: intermediate $>$
S1: The next tram from Marklandsgatan to Brunnsparken is at 2.23pm
U2: Okay
U2′: No, I asked about the next tram to Brunnsgatan

If the recognition score is even lower a more explicit grounding process may be used:

(21) $<$ recognition score for $U1$: low $>$
S1: Did you ask about the next tram from Marklandsgatan to Brunnsparken?
U2: Yes
U2′: No, to Brunnsgatan

A final issue that needs to be addressed in relation to the necessary extensions to the basic GoDiS information state for multimodal generation concerns information structure. Current GoDiS applications do not make use of theories of information structure, and the information state must therefore be extended with information that enables the determination of information structure.

A corpus study carried out as part of the TALK project, and also discussed extensively by [Eri05], has shown a number of different contextual components that need to be kept track of in the information state for the determination of the information structure of an utterance to be generated. The corpus study is a manual analysis of a number of different dialogues from six corpora in English, French, and Swedish, and representing a number of different activities or domains.[4] The analysis is mainly constrained to question-answer dialogues in task-oriented domains, of which dialogues in the tram information domain is another example.

We will here give a review of the main components that were identified as necessary for the determination of information structure in such dialogues. The components are illustrated using dialogue extracts from the corpora. In these extracts, a focus is shown using boldface, and ground using italics. Note that several of the utterances have no ground. If a part of the dialogue is underlined, this is that part of which the ground is a reflection, or which acts as the contextual anchor when the utterance contains only the focus. The underlined portion may be called the *base*, following [Val01]. All corpus examples have been edited into a uniform format to increase readability.

The first component is a question that is locally under discussion at the given point in the dialogue (a *primary* question under discussion), where this question has been explicitly introduced as a question in the dialogue:

[4] For details of the corpora and the dialogues used, see Appendix A of [Eri05].

(22) Component:

A primary question under discussion that has been explicitly introduced

Dialogue example:

Follower: What should I have done?[5]

Giver: *You should have* **gone to the left**

In the dialogue example in (22) the information structure of the giver's utterance, with "You should have" as the ground and "gone to the left" as the focus, is determined using the question that is locally the one that is under discussion, as introduced by the follower in the immediately preceding utterance.

The component in (22) corresponds to the topmost question in QUD in the basic GoDiS information state, and is therefore already in place.

The next component is domain and situation knowledge, which is typically also already part of the information state, in the form of an interface to the domain resource, and such information as who the speakers are (user and system):

(23) Component:

Domain and situation knowledge

Dialogue example:

Customer: **Le programme des films**[6]

Eng. **The film programme**

The utterance in (23) is the first utterance in the dialogue. This means that the hearer of this utterance, in this case the official at the tourist information office, cannot make use of preceding utterances to determine a possible base for this utterance. Instead, domain and situation knowledge is made use of to determine that the customer is probably requesting the hearer to give her the film programme, or tell her where one can be found.

Another component consists of previous questions, which are questions that have been removed from QUD:

(24) Component:

A previous question, that is, a question that has been but is no longer under discussion

Dialogue example:

Travel agent: Hur mår din lilla Josefin?[7]

[5] This example is taken from the HCRC Map Task corpus, dialogue q7nc3. http://www.hcrc.ed.ac.uk/maptask/

[6] This dialogue extract is taken from the Office du Tourisme de Grenoble corpus, dialogue 1AP0228. http://www.sir.blois.univ-tours.fr/~antoine/parole_publique/OTG/

[7] These utterances are taken from the Göteborg Spoken Language Corpus, dialogue A8207051. http://www.ling.gu.se/projekt/tal/

Eng. How is your little Josefin?
Customer: Hon mår hyfsat
Eng. She's okay
Travel agent: Och **Gabriella**?
Eng. And **Gabriella**?

The travel agent's second utterance in (24) has an information structure that is given by her first utterance in the example. When the second travel agent utterance is being produced, the question of Josefin's health has been answered, and is no longer a question under discussion. The component in (24) has no representation in the basic GODIS information state, and will therefore need to be added.

A final component to be discussed here contains answers to previous questions:

(25) Component:
An answer to a previous question

Dialogue example:
Travel agent: And then you're going to return from Chicago on the fourteenth or?[8]
Customer: On the fourteenth right
Travel agent: **At what time**?

The travel agent's second question has an information structure that is determined in relation to the full focus-ground content of the customer's preceding answer, which may be paraphrased as "I'm going to return from Chicago on the fourteenth (right)", making the full proto-content of the travel agent's second question paraphrasable as "At what time are you going to return from Chicago on the fourteenth?".[9]

Answers to previous questions are already represented in the GODIS information state, in the form of the set of propositions that constitute the shared commitments. However, all dialogue utterances that are used for the determination of the information structure of an utterance, when that utterance realises only the focus and no ground, are shown by the corpus study to obey a strong recency constraint. That is, only utterances in the dialogue history that were uttered at most a few turns ago, can be used for the determination of the information structure of an utterance realising only the focus. Judging from corpus examples, it seems that intervening utterances typically make up at most one or two turns, and concern clarification, or grounding more generally.

For utterances consisting of both ground and focus, the recency constraint may not be as strong. However, a structure among preceding utterances is still needed, for the determination of which material is more recent than other material, and this is lacking among the shared commitments in the basic GODIS information state, as these are represented in the form of a set. A structured representation reflecting the order of the dialogue history is also needed for previous questions, that is, for the component in (24).

[8]This is a dialogue extract taken from SRI's Amex Travel Agent Data, tape 1 call 1. http://www.ai.sri.com/∼communic/amex/amex.html

[9]For a motivation of why the information structure of the travel agent's second question in (25) is determined in relation to the customer's answer and not to the travel agent's preceding question, see [Eri05] pages 122-123.

4.2 Representing the extended information state

Turning now to the actual representation of the extended information state, we will discuss this in accordance with the items identified in section 4.1.3 above. The first necessary extension that we identified, is a representation of just which modalities are available. Another necessary addition is the extent to which a given message is to be realised in any one particular modality, for all available modalities. We have added the following two extensions to GODIS as new module interface variables of a new type called Infoamount:

(26) VOICE : Infoamount
GRAPH : Infoamount

The presence of the two variables VOICE and GRAPH in the information state indicates that these are the two possible (output) modalities for the application in question. The values of these variables then indicate the extent to which a particular content is to be realised in the different modalities. The possible values of the variables, that is, the values of the type Infoamount, are given in figure 4.4.[10]

Value	**Explanation**
no	No material at all is (to be) realised in the given modality
min	Only a minimal amount of material, taken to correspond to the focus
interm	An intermediate amount of material, corresponding to the focus and a partial ground
max	A maximal amount of material, corresponding to the focus and a full ground
ground	Only the ground
compl	Complementary, i.e., whatever is not realised in the other modality/ies
indet	The values is indeterminate (unknown)

Figure 4.4: Values of the new type Infoamount

As an example, the system contribution in example (15) above, where the full message is realised both through speech and graphically, may be the result of the following setting:

(27) VOICE : max
GRAPH : max

Similarly, example (16), where the focus is realised through speech and the whole message graphically, may be determined from:

[10] Partial grounds, mentioned in relation to the value 'interm', are not discussed in the present chapter, but are included in the implementation for future extensions. For an empirically based discussion of partial grounds, see [Eri05].

(28) VOICE : min
GRAPH : max

In this way, the values in figure 4.4 give a number of different ways of specifying all the different possible contextualisations for a given proto-content. As another example, the contribution in example (17) involves the ground being spoken and the focus shown on the map. This may, for instance, result from GRAPH having the value 'min', and VOICE either having the value 'ground' or 'compl'.

To enable the representation of the actual output to be produced by the system, GODIS, as explained in section 4.1.1 above, contains a module interface variable OUTPUT which takes a value of type string. This variable then records the output to be given through speech (or through written text). Another MIV is then needed for the representation of the actual output to be given to the graphical output, which gives another extension to the total information state:

(29) GRAPH_OUTPUT : String

A final new MIV that is here added to the information state, is needed for dealing with communication difficulties as signalled by the speech recognition score. The actual speech recognition score is already recorded in the total information state through the MIV SCORE (see figure 4.2 on page 44), which takes a value in the form of a real number. We make use of this number to determine the value of the new MIV SPEECHINPUT:

(30) SPEECHINPUT : Infoamount

This new MIV takes a value of type Infoamount, and the value is determined by the value of the SCORE variable, as we will see below.

The addition of the MIV SPEECHINPUT means that there may be a conflict between different sources as to how much information is to be realised. Consider the following setting:

(31) VOICE : no
GRAPH : min
SPEECHINPUT : max

The values of VOICE and GRAPH indicate that a minimal amount of information, corresponding to the focus, is to be realised graphically, and no part of the message is to be realised through speech. However, the value of SPEECHINPUT indicates that both the focus and the ground need to be realised, presumably as a result of the speech recognition score being too low to warrant only the focus being realised. Such conflicts are the topic of other work carried out as part of work package three, but not discussed here. It is quite clear, however, that for example (31), the value of SPEECHINPUT needs to take precedence, so that ground material is also realised.

The default value of SPEECHINPUT at the start of a new dialogue, before any utterances, is 'indet'.

The final extension to the information state is the addition of a structured representation of recent questions and answers, which is needed for the determination of the information structure of a system contribution to be generated. This will be represented in the information state in the form of a dialogue move history in the shared part of the information state. This move history keeps track of recent user and system contributions. Each contribution is here to be seen as an open queue of dialogue moves (note that an utterance or contribution may consist of several moves), and the dialogue move history as an open queue

of such open queues of moves.[11] The most recent contribution is then the last oqueue of moves in the move history, and a structure is thus enforced on recent contributions.

A representation of the fully extended GoDiS information state is given in figure 4.5 on page 54. This also includes the module interface variables which are part of the total information state.

4.3 Maintaining the information state during dialogue

Having represented the necessary extensions in the GoDiS information state, we will now detail how this extended (total) information state is maintained during the dialogue, that is, how the values of the extensions are dynamically updated.

Beginning with the values of the available modalities VOICE and GRAPH, we noted in the discussion in section 4.1.3 a need for default values, that is, a need for default behaviour with respect to the distribution of a system contribution across modalities. At the start of an interaction with GoDiS, the user types `run`, and a number of properties are then automatically set before the actual dialogue starts. As part of this, VOICE and GRAPH can be set to default values. Currently these are 'min' and 'compl' respectively, but they can of course be set to any value of the type Infoamount.

In addition to this, in the current implementation we also give the user the option of starting the programme in a particular mode, by typing a different run command. We have illustrated this by giving four different modes:

(32) `rundriving` starts the programme and sets VOICE to 'max' and GRAPH to 'no'. This handles a situation where the user is not to take her eyes off the road to look at a screen while driving, and where she is to be given highly explicit spoken information so that no unnecessary confusion impedes the driving.

`runtelephone` starts the programme and sets VOICE to 'min' and GRAPH to 'no'. This handles a situation where the user is unable to see a screen, but can be met with focus-only information through speech, to create an efficient dialogue.

`runmeeting` starts the programme and sets VOICE to 'no' and GRAPH to 'max'. This handles a situation where the user needs to be quiet and hence cannot use speech but only information conveyed graphically.

`runathome` starts the programme and sets VOICE to 'min' and GRAPH to 'max'. This handles a situation where the user has access to both a screen and spoken output, such as at home in front of a computer.

Note that these different modes are to be seen as an illustration of the management of the information state. The modes can easily be changed to involve other values if motivated by empirical data or otherwise desirable, and it is also possible to define other modes as part of the design of the system.

The values of VOICE and GRAPH cannot only be changed at the start of the dialogue, but also during the dialogue through user input. This enables the user to change the values one or several times as part of an ongoing interaction. One way that we have implemented this is by allowing the user to choose one of the predefined modes during the dialogue. Consider the following interaction with the system (note that an

[11] An open queue is one of the data types already in place in GoDiS. Another possibility may be to make use of a new datatype "limostack", that is, an open stack of a limited length. We may pursue this possibility in future work.

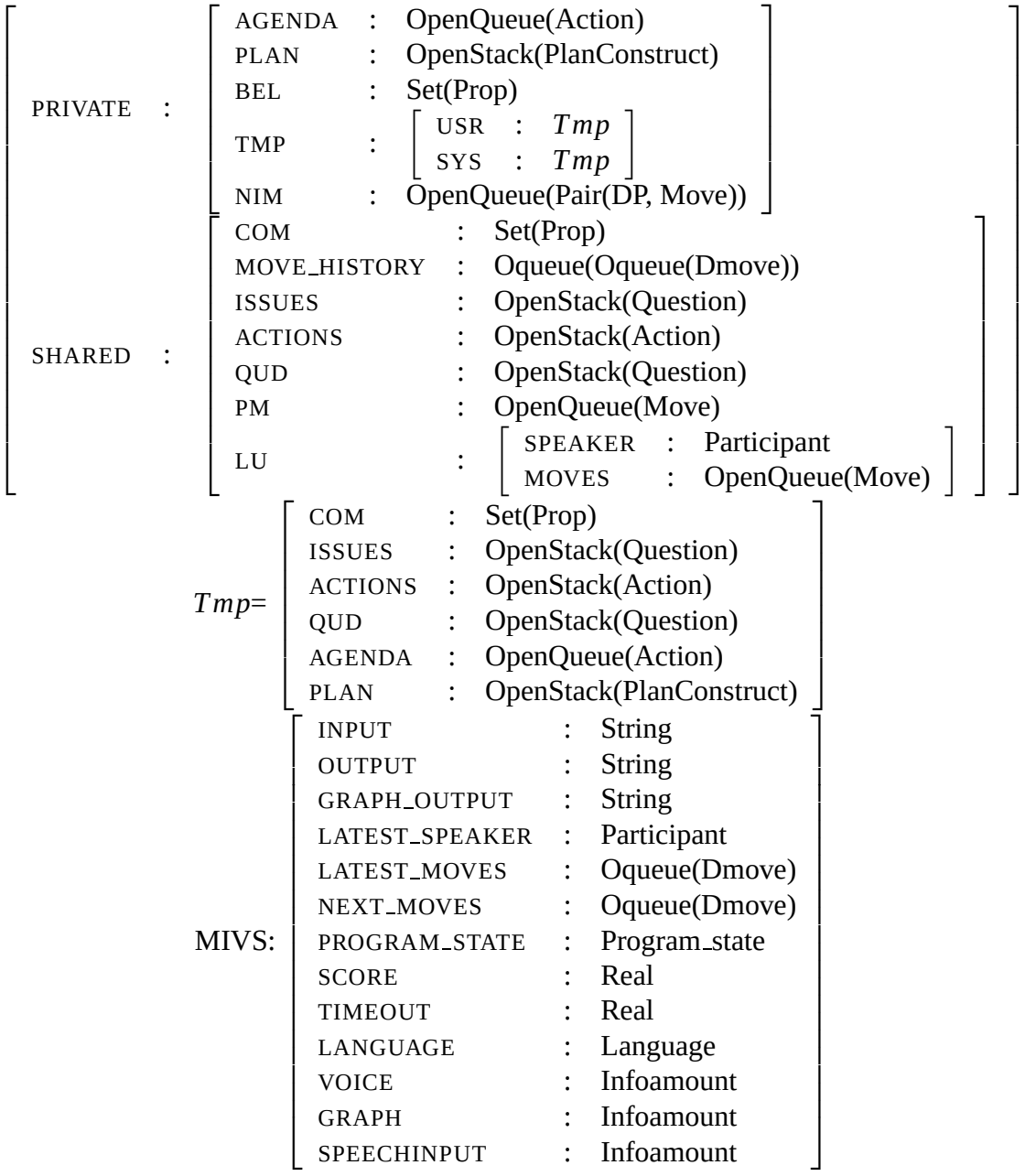

Figure 4.5: The extended GODIS information state and module interface variables

empty user contribution following a system *wh*-question will here result in an alternative question from the system, explicitly showing the user the available options):

(33) S> Welcome to the Tram information system!
S> Let's see. What can I do for you?
U>
S> Do you want to have information about tram times or set the output mode of the system?
U> Set the output mode
S> Okay. Let's see. What do you want to do concerning the output mode of the system?
U>
S> Do you want to set the speech output, set the graphical output, set both speech and graphical output or use one of the predefined modes?
U> Predefined
S> Okay. Let's see. Which predefined mode do you want to use?
U>
S> Do you want to use "Driving" , "Telephone" , "Meeting" or "At Home"?
U> Driving
< VOICE is set to 'max' and GRAPH is set to 'no' >
S> Okay.

Here the user is able to choose between one of the predefined modes as part of the dialogue. Conditions may for instance change during an interaction with the system, so that the user wants to be able to change the mode without having to restart the system.
A much shorter exchange is also allowed by the GODIS mechanism of question accommodation:

(34) S> Welcome to the Tram information system!
S> Let's see. What can I do for you?
U> Driving
< VOICE is set to 'max' and GRAPH is set to 'no' >
S> Okay.

Finally, the values of VOICE and GRAPH can also be set individually, and to specific values. This is done through dialogue, as in the following example:

(35) ...
S> Do you want to have information about tram times or set the output mode of the system?
U> I want to set the output mode of the system
S> Okay. Let's see. What do you want to do concerning the output mode of the system?
U>
S> Do you want to set the speech output, set the graphical output, set both speech and graphical output or use one of the predefined modes?
U> Set both speech and graphical output

S> Okay. Let's see. What value do you want to set the speech output to?

U>

S> Do you want to set the speech output mode to "nothing", "minimum", "maximum", "ground", "complementary", or to "intermediate"?

U> Ground

< VOICE is set to 'ground' >

S> Okay. What value do you want to set the graphical output to?

U> Maximum

< GRAPH is set to 'max' >

S> Okay. What else can I do for you?

Let us turn next to another new module interface variable, SPEECHINPUT. This variable records the degree to which a given message needs to be realised in the *modalities taken together*, based on the recognition score of the latest user input (see the discussion above). The value of SPEECHINPUT at any given point during the dialogue is therefore determined by the speech recognition score. We have implemented this as part of the update rule **getLatestMoves**, which sets the value of SPEAKER and MOVES in LATEST UTTERANCE, after a new utterance in the dialogue. In addition, it now sets SPEECHINPUT to 'min' if the recognition score is above 0.9 (the maximal value being 1), and otherwise to 'max'. This is intended to reflect that all output modalities must together realise (at least) the focus information (corresponding to 'min') if the recognition is very good, but not necessarily any ground material. However, if the recognition score is lower, the full ground material must also be included in at least one of the modalities (corresponding to 'max'). The value of 0.9 can be changed to any other value should this seem more appropriate.

There is no possibility for the user to set the value of SPEECHINPUT using dialogue, as this value is determined in relation to the speech recogniser. However, when running the system in a text-input mode, the value of the recognition score is manually set by the user to simulate the behaviour of the speech recogniser. Compare, for instance, the following two dialogue examples (the prompt asking for the recognition score was removed from dialogues (33), (34), and (35) above to increase readability):

(36) S> Welcome to the Tram information system!

S> Let's see. What can I do for you?

U>

[score 0.0-1.0]:

S> Do you want to have information about tram times or set the output mode of the system?

U> Tram times

[score 0.0-1.0]:

S> Okay. Let's see. What do you want to know?

U> What time is the next tram from Marklandsgatan to Brunnsparken

[score 0.0-1.0]: 0.67

< SPEECHINPUT is set to 'max' >

(37) S> Welcome to the Tram information system!

S> Let's see. What can I do for you?

U>
[score 0.0-1.0]:
S> Do you want to have information about tram times or set the output mode of the system?
U> Tram times
[score 0.0-1.0]:
S> Okay. Let's see. What do you want to know?
U> What time is the next tram from Marklandsgatan to Brunnsparken
[score 0.0-1.0]: 0.92
< SPEECHINPUT is set to 'min' >

In dialogue (36) giving the recognition score as 0.67 sets the value of SPEECHINPUT to 'max', indicating that the ensuing system contribution should contain both focus and ground. In dialogue (37), on the other hand, 0.92 sets the value of SPEECHINPUT to 'min', which means that the system is sufficiently certain of having understood the user correctly to convey only focus material through its next contribution. The default value for the recognition score in text mode is 1.0, so that pressing return without typing in any value for the score, as in the first two user utterances in both dialogue (36) and (37), gives the value 'min' to SPEECHINPUT.

The final new module interface variable, GRAPH_OUTPUT, contains the information that is to be given to the graphical output device. As the precise content of this concerns work carried out elsewhere in this work package, it is not reported on here.

The final extension to the information state that we made above was the addition of a structured dialogue history, in the form of an open queue where the current latest moves are always added. This structure is updated by another addition to the update rule **getLatestMoves** mentioned above. Figure 4.6 shows the parts of **getLatestMoves** that are of particular relevance to the present discussion. Some of the preconditions and the effects have been removed, as indicated by the dots.

The preconditions in **getLatestMoves** shown in figure 4.6 gets the values of the three MIVs LATEST_MOVES, LATEST_SPEAKER, and SCORE, and assigns these to variables through unification. The push operation in the effects adds LatestMovesQueue, corresponding to the moves of the latest turn in the dialogue, to the MOVE_HISTORY queue, as the last element in that queue.

However, as I discussed above, there is a recency constraint on which contributions in the dialogue history can be used for the determination of the information structure of a contribution to be generated. It is therefore unnecessary to keep the whole dialogue history in the information state, and a limit on the number of contributions in the dialogue history can therefore be used. The update rule **getLatestMoves** has been modified to include such a limit, as can be seen from the two lines above the push operation. The limit is here arbitrarily set to 6, meaning that whenever the move history contains moves corresponding to the six most recent contributions, the oldest contribution is removed before the addition of a new contribution through the push operation.

Figure 4.6 also shows the effects corresponding to the determination of the MIV SPEECHINPUT discussed above.

```
RULE: getLatestMoves
CLASS: grounding
PRE:
    $latest_moves = LatestMovesQueue
    $latest_speaker = DP
    $score = Score
    ...
EFF:
    ...
    if_do( $$arity( $/shared/move_history ) == 6
        pop( /shared/move_history ))
    push( /shared/move_history, LatestMovesQueue )
    if_do( DP = usr,
        if_then_else( Score > 0.9,
                speechinput := min,
                speechinput := max ) )
```

Figure 4.6: Parts of the update rule **getLatestMoves**

4.4 Summary and conclusion

This chapter has reviewed a number of relevant aspects of the GoDiS system as used in a number of different unimodal applications. These aspects include the different fields in the information state, the interface variables that are part of the total information state, and update rules that maintain the information state during dialogue. The chapter then discussed a number of different factors needed for multimodal generation in GoDiS. These include information about what modalities are available, how much material is to be realised in any given modality at a particular point, evidence from the speech recogniser that influences whether content reduction can be used or not, and a number of different aspects of the linguistic and non-linguistic context needed for the determination of information structure.

The chapter continued by definining representations of necessary extensions in the information state, viz. the module interface variables VOICE, GRAPH, and SPEECHINPUT, and the different values these can take, in the form of Infoamount values. The information state was also extended with a MOVE_HISTORY, giving a structured representation of preceding utterances as needed for the determination of information structure. Finally, the chapter developed methods for the maintenance of the extended information state in the form of default modes to be used when starting the system, or during a dialogue, as well as the possibility of setting values explicitly and one by one. These methods also include new and modified update rules for the values of the three new module interface variables, and the MOVE_HISTORY.

In conclusion, starting from a number of possible scenarios this chapter has shown in what ways the GoDiS information state needs to be extended for the appropriate generation of multimodal contributions, and it has provided an implementation of this (see also the Appendix). The existence of a central repository of information in the form of the information state has shown that it is straightforward to add capabilities for the determination of multimodal contributions to a unimodal system, and the information state provides a theoretically attractive model of the material needed for multimodal dialogue.

Chapter 5

The Extended Information State in the UEDIN/UCAM In-Car System

The UEDIN/UCAM baseline in-car dialogue system was constructed primarily in order to be able to collect data for Reinforcement Learning (RL) approaches to multimodal dialogue management, and also to test and further develop learnt dialogue strategies in a realistic application scenario. The extensions to the IS made by UEDIN for the UEDIN/UCAM system focus on extensions for reinforcement learning (namely, the IS fields for task state, dialogue history, and reward) and for fragmentary clarifications (namely the IS field for word-based confidence scores).

In order to support reinforcement learning (RL), the IS needed to be extended with history features (so that learning has access to dialogue history), task features (so that learning can take account of task completion), and rewards (so that the RL algorithm has an appropriate reward signal).

The other extension made was an IS field for word-based confidence scores (from the ATK recognizer). These are used to generate fragmentary clarification requests from the system (see D4.2).

In this chapter we present and motivate the IS definition used in this system, give examples of update rules used to maintain these IS, and describe how the states can be extended to include multimodal information.

5.1 IS structure: the UEDIN/UCAM IS definition

Figure 5.1 shows the Information State (IS) definition currently used in the UEDIN/UCAM in-car system (see TALK deliverable D4.2 [LGS05]). Other aspects of these states are presented in the publications [GHL05, HLG05, GLH05].

5.1.1 Explanation of low level, dialogue level, history level, and reward level of the IS

The IS defined above can be described using 4 main levels:

- dialogue-level
- task-level

```
infostate(record([is:record([
           flglearn:atomic,  % System parameter: learning on or off

                          %%%% DIALOGUE level features
           turn:atomic,
           speaker:atomic,
           lastspeaker:atomic,
           input:stack(atomic),
           lastinput:stack(atomic),
           output:stack(atomic),
           nextmoves:stack(Acts),
           cursysmove:stack(Acts),
           confirmnumber:atomic,
           prompttype:atomic,
           speechact:stack(atomic),

                               %%%% LOW level features
           confidence:atomic,
           confidenceword:atomic,
           implicitconf:atomic,
           ngram:atomic,
           lastinputconf:stack(atomic),
           lastinputconfuttered:stack(atomic),

                       %%%% TASK level features

           task:stack(atomic),
           subtask:stack(atomic),
           taskstep:atomic,
           taskstepname:atomic,
           filledslotsvaluesshow:stack(atomic),
           filledslotsshow:stack(atomic),
           filledslotsvalues:stack(atomic),
           filledslots:stack(atomic),
           confirmedslots:stack(atomic),
           confirmedslotsvalues: stack(atomic),

                          %%%% HISTORY level features
           numinformationstate:atomic,
           lastmoves:stack(Acts),
           speechactshist:stack(atomic),
           subtaskshist:stack(atomic),
           filledslotsvalueshist:stack(atomic),
           filledslotshist:stack(atomic),
           confirmedslotshist:stack(atomic)])]).
```

Figure 5.1: The UEDIN/UCAM in-car system Information State definition

- low-level
- history-level

Dialogue-level features

For each utterance we track the following features at the "dialogue level" of the IS, which corresponds to information about the most recent (user or system) utterance:

- turn: the agent (user/system) who has the turn
- speaker: the agent (user/system) who performed the current utterance
- lastspeaker: the agent (user/system) who performed the previous utterance
- input: the recognized string of the current utterance (if any)
- lastinput: the recognized string of the preceding utterance (if any)
- output: the string of the system output (if any)
- nextmoves: the next planned moves by the system (if any)
- cursysmove: the current system move (if any)
- confirmnumber: how mant times the system has tried to confirm the current slot
- prompttype: whether the prompt is optional or mandatory for the user to answer
- speechact: the semantics of the current utterance (see extended DATE scheme)

Task-level features

For each utterance we also track the following information about the task level of the IS:

- task: name of the current domain task
- subtask: name of the current subtask (if any)
- taskstep: step number in the current task
- taskstepname: step name in the current task
- filledslotsvaluesshow: current values of the filled slots shown on the GUI
- filledslotsshow: names of the currently filled slots shown on the GUI
- filledslotsvalues: current values of the filled slots
- filledslots: names of the currently filled slots
- confirmedslots: names of the currently confirmed slots
- confirmedslotsvalues: current values of the confirmed slots

Low-level features

For each utterance we use the IS to track the following low-level features:

- confidence: confidence score for current input
- confidenceword: word-based confidence scores for current input

History-level features

The history-level annotations/logs of states are be computable from the above information over the history of utterances. As an initial set of history-level features, for each state of the dialogue we use the following:

- numinformationstate: a count of the number of IS there have been in the dialogue thus far
- lastmoves: sequence of all the preceding moves in the dialogue
- speechactshist: sequence of all the preceding speech acts in the dialogue
- subtaskshist: sequence of all the preceding tasks/subtasks in the dialogue
- filledslotsvalueshist: sequence of all the preceding filled slot values in the dialogue
- filledslotshist: sequence of all the preceding filled slot types in the dialogue
- confirmedslotshist: sequence of all the preceding confirmed slot types in the dialogue
- confirmedslotsvaluueshist: sequence of all the preceding confirmed slot values in the dialogue

Reward level features

As an important part of IS, we also collect reward-level features for each dialogue. Reward level features are currently only assigned to final IS.

The reward features for each dialogue may be gathered either by online questionnaires, paper questionnaires, or could be be computed from individual user actions (e.g. hang up). We currently use the following:

- task completion (actual and perceived)
- dialogue duration
- number of turns
- user satisfaction
- PARADISE evaluation metrics (see [WKL00] for examples)

Note that these IS features are used to learn dialogue strategies [LGH$^+$05, LGS05].

5.1.2 Methods for maintaining the IS

We maintain these IS by means of update rules, expressed in the update language of the DIPPER ISU dialogue management [BKLO03b] system.

DIPPER is similar to Trindi [LT00], taking its record structure and datatypes to define information states (see the example information state in figure 5.1). In contrast to TrindiKit, in DIPPER all control is conveyed by update rules, there are no additional update and control algorithms, and there is no separation between update and selection rules. Furthermore, the update rules in DIPPER do not rely on Prolog unification and backtracking, and allow developers to use OAA "solvables" in their effects (see [LGS05] for examples).

Update rules (see examples in figures 5.2 and 5.3) specify information state change potential in a declarative way: applying an update rule to an information state results in a new state. An update rule is a triple $<$ `name, conditions, effects` $>$. The conditions and effects are defined by an update language, and both are recursively defined over terms. The terms of the update language allow developers to refer to specific values within information states, either for testing a condition, or for applying an effect. For example, the term `isˆlastspeaker` refers to the the field `lastspeaker` in the record `is`.

In the update rules shown in figures 5.2 and 5.3 `solve2` is used to request OAA services. For example:

```
solve2(text-to-speech(isˆoutput))
```

sends a request via OAA for a "text-to-speech" task with 1 argument (the value of the "output" field in the current IS `is`). In the current system, such requests are handled by the Festival OAA agent, but they could be handled by any other agent which registers the text-to-speech solveable with the OAA facilitator (the hub).

Figure 5.3 shows an update rule which initialises the Information State features, the ATK speech recogniser, the map GUI, and the file where the logging agent stores all the information states of the dialogue.

Figure 5.3 shows an example update rule which triggers explicit confirmation requests from the system, and how this maintains the IS. It is triggered when the ASR confidence is lower than a threshold, and there have been fewer than 3 explicit confirmation attempts on the slot value. The main effects are to generate an explicit confirmation request and maintain the speech act history in the IS.

5.1.3 Adding multimodality

The current UEDIN/UCAM in-car system allows system graphical output (on the map GUI), but does not yet process multimodal user inputs. In future we expect to enhance the IS and update rules to deal with multimodal input as well as output (there are already update rules which display items on the system's map GUI).

Here we outline the extensions which can be made to the existing IS to incorporate these aspects of multimodality.

Multimodal IS features

- available input modalities
- available output modalities

```
urule(initialisation,
     [
      is^lastspeaker=''     %%% CONDITIONS (=start of system)
     ],
     [                      %%%% EFFECTS: initialize IS and OAA agents

% initialise GUI with SACTI map
    solve(displayAgent_change_background_image('/talk/incar/guiagent/sactiMap.gif',
            ResDisplay)),

% ATK speech recogniser initialisation
    solve2(atkinitrecogniser('ATKrecognisercfg','general_diction_ucase',
           'yesno_lattice_ucase_o,wordloopnew',Res)), % Result 0 is ok, 1 problem
    assign(is^recognflag, Res),
     prolog(write('ATK initialized')),

% open file for logging
    solve2(openissequencefile('/group/project/talk/incar/incar_log.txt',ResTxt)),
    assign(is^txtflag, ResTxt),

% initialize Information State for start of dialogue
      assign(is^lastspeaker, user),
      assign(is^turn, system),
      push(is^task, 'task'),
      assign(is^taskstepname, 'greet'),
      assign(is^taskstep, 0),
      push(is^lastinputconf,[null]),
      push(is^lastinputconfuttered,[null]),
      push(is^cursysmove,null)
     ]
    ).
```

Figure 5.2: IS Initialisation Update rule from the UEDIN/UCAM in-car system

```
urule(generation_confirmation_explicit,
     [                        %%% CONDITIONS
      is^confirmnumber\=3,
      is^confidence=1, % i.e. ASR conf lower than confirmation threshold
      is^lastspeaker=user,
      prolog(check_slot('([quit_request],u)',top(is^filledslots),ResSlot)),
      ResSlot=0,
      prolog(check_slot('([restart_request],u)',top(is^filledslots),ResSlot1)),
      ResSlot1=0,
      prolog(check_slot('([ask_happy],s)',top(is^filledslots),ResSlot2)),
      ResSlot2=0,
      is^flgsave=0
     ],
     [                        %%% EFFECTS
% increment the counter
      prolog(increase_num(is^confirmnumber,ConfNumber)),
      assign(is^confirmnumber,ConfNumber),
% generate explict_confirm prompt (TextOut)
      prolog(utter_prompt('([explicit_confirm],s)',top(is^lastinputconfuttered),TextOut,
                          Type,ValueYes,ValueNo,SpeechAct,top(is^task),is^taskstepname)),
% say TextOut via Festival TTS
      solve2(callfestival('/festival/bin',TextOut,_Y)),
% update Information State
      assign(is^confidence,0),
      push(is^lastmoves,['([explicit_confirm],s)']),
      clear(is^subtask),
      push(is^subtask, is^taskstepname),
      clear(is^speechact),
      push(is^speechact, SpeechAct),
      push(is^subtaskshist, top(is^subtask)),
      push(is^speechactshist, top(is^speechact)),
      clear(is^output),
      push(is^output,TextOut),
      assign(is^prompttype,Type),
      assign(is^lastspeaker,system),
      clear(is^input),
      assign(is^turn,user)
     ]
    ).
```

Figure 5.3: IS maintenance: explicit confirmation update rule from the UEDIN/UCAM in-car system

- input modalities used in current user input
- output modalities used in current system output
- XY co-ordinates of mouse gestures
- n-best list of reference hypotheses (e.g. for clicked objects)
- system output graphical reference objects
- last-used system modalities
- last-used user modalities
- sequence of all user graphically-referenced objects
- sequence of all system graphically-referenced objects

5.2 Conclusion

In this chapter we have presented and explained the IS definition used in the current UEDIN/UCAM in-car dialogue system. We also gave examples of update rules used to maintain the IS, and described how the states can be extended to include multimodal information. The prototype included in the appendix contains the current implementation of the extended information state within a preliminary version of the UEDIN/UCAM final in-car showcase.

Chapter 6

The DELFOS Extended Information State

6.1 Introduction

This chapter describes the evolution of the Delfos Information State in the TALK project. We first discuss the overall goal of USE within Task 3.1. Then we describe the original Delfos DTAC structure. The original DTAC information state has been extended with two purposes in mind: (a) to allow for multimodal input and output, and (b) to allow for a direct translation of DTAC structures to DB queries. We discuss the motivation in more detail and describe the new extensions. Finally, some conclusions will be drawn and future lines of work will be described.

The work to be discussed in this chapter is intimately related to the on–going work being carried out in WP1. Therefore, and in order to be coherent, the presentation of information will also be defined within a scale of modes ranging from speech–only output, to graphics–only output.

Nonetheless, there will be no direct and unique correspondence between speech–only input and speech–only output, or between graphics–only input and graphics–only output. It is easy to think of situations where for instance, all the input would normally be spoken (speech–only) and all the output would be graphical[1]. Let us then consider the full range of possibilities. Input and output modes will therefore be determined in relation with the situational context, the type of amount of information being exchanged and the users needs or wishes[2].

Regarding multimodal output, USE has focused on three modalities:

- Speech output: The system will speak to the user using natural language.
- Graphical output: The system will output written text, images or both, depending on functionality and context.
- Mixed output: The system will be able to both speak and display images and/or text simultaneously.

As noted when discussing the possible input modalities, it is interesting to highlight the fact that co–occurring modalities will not necessarily be complementary, that is, we intend to contemplate complex

[1] Consider, for example, a command with which the user requires a great deal of information, or the display of a map, etc. The users input would likely be spoken, whereas the systems output would necessarily be graphical.

[2] We are especially sensitive to users with disabilities who may benefit from this research.

situations where users choose to multitask and therefore co–occurring events (inputs or outputs) may be completely unrelated. This issues will be discussed on D3.2.

6.2 The Delfos DTAC Structures

This section describes how the Information State Update approach to dialogue management is interpreted and implemented in Delfos.

In Delfos, the main part of the information state is a dialogue history, represented formally as a list of Dialogue States. Dialogue rules update this structure either by producing new Dialogue States or by supplying arguments to existing Dialogue States.

An information state theory of dialogue modeling consists of:

- a description of the **informational components** of the theory of dialogue modeling
- **formal representations** of such informational components
- a set of **dialogue moves** that will trigger the update of the information state
- a set of **update rules** that govern the updating of the information state
- an **update strategy** to decide which rule(s) to apply at a given point

The informational component

In the first place, it must be decided whether the participants' internal state should be modeled, or we should rather focus on more external aspects of dialogue.

As a consequence of their own nature, Natural Command Language Dialogues involve just two participants: user and system. Since the goal of this type of dialogues is to provide the user with full control over the execution of one or more commands by the system, most dialogues exhibit a noticeable functional or operational tendency. Additionally, the sequence of commands may sometimes lack logical flow, or even be the consequence of unclear intentions during the dialogue. It is for all this that it seems reasonable to focus our model on the external aspects of dialogue. That is, it should be based more on what was said than what was in the minds of the participants when things were said.

In the second place, the information state could be defined as a dynamic or static entity. Since one of the features which distinguish Natural Command Language dialogues from Information Seeking dialogues is the dynamic nature of the knowledge bases involved, the dynamic nature of the NCL information state is quite clear.

In an Information Seeking dialogue, the system as a whole is viewed by the user as a repository of knowledge; knowledge bases have a clear static character from the users point of view. That is, the data stored in these resources may be updated, but not by the user during its interaction with the system.

On the contrary, one of the main features of Natural Command Language dialogues is the presence of a command execution system (an Action Manager in the Delfos agent architecture) which is capable of dynamically modifying the contents of external resources to the dialogue, such as the knowledge bases associated with the domain (by means of the Knowledge Manager).

The formal representation

The USE information state is represented as a feature structure (also called a DTAC structure) containing four attributes: Dialogue Move, Type, Arguments and Contents. More attributes may be added in the course of the dialogue update, as for example the expectations (EXPT) generated by each dialogue rule.

1. **DMOVE**: This feature identifies the kind of dialogue move. Its range of values is the set of dialogue moves.

2. **TYPE**: This feature identifies the specific dialogue move type within the kind of the corresponding DMOVE. For instance, the utterance "*Could you please turn on the kitchen light.*" belongs to the *specifyCommand* DMOVE category, and to the *SwitchOn*TYPE in the Home domain application.

   While the DMOVE classification intends to be domain and implementation independent, the set of TYPEs is domain dependent. In some sense, the TYPE classification instantiates the DMOVE model to the specific domain.

3. **ARGS**: Some types of dialogue moves may require the presence of one or more arguments. The ARGS feature specifies the argument structure of the DMOVE/TYPE pair. This takes the form of a list in which conjunction, disjunction, and optional operators may appear.

4. **CONT**: This feature represents the particular values associated with each element in the ARGS attribute.

   For terminal DTAC structures (with an empty ARGS list), the CONT feature will specify the value of the structure.

   For non–terminal DTAC structures (with a non–empty ARGS list), CONT is recursively represented by the CONT feature of each feature whose name equals the ARGS value.

As an illustration, the following is DTAC represents the utterance *llama a luis (Call luis)*.

$$\begin{bmatrix} \text{DMOVE} & \text{specifyCommand} \\ \text{TYPE} & \text{MakeCall} \\ \text{ARGS} & \begin{bmatrix} \text{DEST} \end{bmatrix} \\ \text{DEST} & \begin{bmatrix} \text{DMOVE} & \text{specifyParameter} \\ \text{TYPE} & \text{Name} \\ \text{CONT} & \text{luis} \end{bmatrix} \end{bmatrix}$$

Dialogue Moves

One of the motivations for this classification is that it is abstract enough to be able to encode the domain–independent aspect of any Natural Command language. This classification has been used in two domain applications: the Home Environment domain under the D'Homme project [3], and the Automatic Telephone Operator system in the Siridus project [4]

[3] www.ling.gu.se/projekt/dhomme/

[4] www.ling.gu.se/projekt/siridus/

	Dialogue Moves (DM) in NCL
Command-oriented DMs	askCommand specifyCommand informExecution
Parameter-oriented DMs	askParameter specifyParameter
Interaction-oriented DMs	askConfirmation answerYN askContinuation askRepeat askHelp answerHelp errorRecovery greet quit

6.3 Extending the DTAC Structure

In order to allow for multimodality the original DTAC structure was extended as it will be described in the following sections. However, additional extensions have been implemented in order to allow for new functionality relevant to the current project. More precisely, the DTAC structure has been extended to include restrictions that will allow for the direct translation of a DTAC to a DB query.

6.3.1 Extending the DTAC Structure to include multimodality

To begin with, two new categories have been added to the basic DTAC structure, in order to allow for multimodality

- `MOD_TYPE`
- `TIME_ST`

`MOD_TYPE:` It corresponds to the "Modality Type" of input or output. Its possible values will depend on the modalities available.

`TIME_ST:` It corresponds to "Time Stamp", and it is currently defined as a time interval that marks the beginning and the end of the utterance / Input.

This information may be more or less granular within the DTAC depending on the information available from the speech recognizer, since not all recognition engines can provide precise information about the start and end points of words or phrases within a longer utterance. In the case of graphical input (clicking), the interval is likely to be equivalent to a point in time. Regarding this type of input, we anticipate some issues related to multiple clicks on the same area within a small time frame, since they are likely to be equivalent to just one click. The interval within which multiple clicks will be equivalent to one unique click will be defined according to experimental data.

This new DTAC structure is valid for both input and output. We could say that when new input is being analysed, these two new pairs describe what has happened, that is, the modality chosen by the user, and the time interval within which the input occurred. However, in the case of the system output, these pairs describe what needs to be generated, that is, the modality chosen by the system to output information, and the time at which it must happen. This latter consideration is particularly important when dealing with mixed–mode output.

Consider the following scenario.

User (speech) (screen)	Turn this on (clicks on a specific light icon in the kitchen on the screen)
System (screen)	(Displays the appropriate icon) (Turns the selected device on)

The speech input generates the following DTAC structure:

$$\begin{bmatrix} \text{DMOVE} & \text{specifyCommand} \\ \text{TYPE} & \text{CommandOn} \\ \text{ARGS} & \left[\text{DEVICESPECIFIER}\right] \\ \text{MOD\_TYPE} & \text{speech} \\ \text{TIME\_ST} & \text{00:00:00-00:00:00} \end{bmatrix}$$

In turn, the click generates the corresponding DTAC as well:

$$\begin{bmatrix} \text{DMOVE} & \text{specifyParameter} \\ \text{TYPE} & \text{DeviceSpecifier} \\ \text{ARGS} & \left[\text{DEVICETYPE},\text{LOCATION},\text{IDENTIFIER}\right] \\ \text{IDENTIFIER} & \begin{bmatrix} \text{DMOVE} & \text{DeviceSpecifier} \\ \text{TYPE} & \text{Identifer} \\ \text{ARGS} & \\ \text{CONT} & \text{kitchen01} \end{bmatrix} \\ \text{MOD\_TYPE} & \text{graphical} \\ \text{TIME\_ST} & \text{00:00:00-00:00:00} \end{bmatrix}$$

As described in Deliverable D2.1, the Knowledge Manager will call the OWL ontology in order to solve the reference of Identifier `kitchen01` above. This call is performed via an RDQL query such as the following:

```
kmGetProp(kitchen01,[locatedIn,DeviceType],)
```

which returns the following result:

```
kmGetProp(kitchen01,[locatedIn,DeviceType],[kitchen,light])
```

that is, `kitchen01` is LocatedIn kitchen and `kitchen01` is of DeviceType light.
As a result, the information state above will be enriched with this information, obtaining a DTAC such as this:

DMOVE	specifyParameter
TYPE	DeviceSpecifier
ARGS	[DEVICETYPE,LOCATION,IDENTIFIER]
DEVICETYPE	DMOVE DeviceSpecifier TYPE DeviceType ARGS CONT light
LOCATION	DMOVE DeviceSpecifier TYPE Location ARGS CONT kitchen
IDENTIFIER	DMOVE DeviceSpecifier TYPE Identifer ARGS CONT kitchen01
MOD_TYPE	graphical
TIME_ST	00:00:00-00:00:00

Next, and following our multimodal fusion strategy described in Deliverable D1.2, both DTACs will be fused by the dialogue manager into a single information state with the following shape:

```
[DMOVE          specifyCommand
 TYPE           CommandOn
 ARGS           [DEVICESPECIFIER]
 MOD_TYPE       speech
 TIME_ST        00:00:00-00:00:00
 DEVICESPECIFIER [DMOVE      specifyParameter
                  TYPE       DeviceSpecifier
                  ARGS       [DEVICETYPE,LOCATION,IDENTIFIER]
                  DEVICETYPE [DMOVE  DeviceSpecifier
                              TYPE   DeviceType
                              ARGS
                              CONT   light]
                  LOCATION   [DMOVE  DeviceSpecifier
                              TYPE   Location
                              ARGS
                              CONT   kitchen]
                  IDENTIFIER [DMOVE  DeviceSpecifier
                              TYPE   Identifer
                              ARGS
                              CONT   kitchen01]
                  MOD_TYPE   graphical
                  TIME_ST    00:00:00-00:00:00]]
```

The current system architecture implies the distribution of a series of agents over several platforms and therefore several independent clocks. Time stamping is therefore not trivial since the time difference between the clocks might have an impact on the decision process if not compensated. The time stamping strategy must therefore take into account this multi–platform distribution. It is for this reason that agents have been modified to provide time information.

6.3.2 Extending the DTAC Structure to include restrictions

Although the former extensions to the DTAC structure described in previous sections are sufficient to deal with multimodality and queries from the Knowledge Manager over OWL ontologies, additional extensions must be made to enable the DTAC to cope with more complex queries; that is, the kind of questions that should be translated into a DB query should the application require it.

One illustrative example of these additional extensions is the MP3 scenario, where the system would have to retrieve information from the MP3 Database. Although the information contained in the MP3 domain could also be represented as an OWL ontology, a DB representation seems more suitable, since there are not many relations among the different objects.

List all of Queens songs between 1981 and 1985

```
[ DMOVE      SpecifyCommand
  TYPE       ListSongs
  ARGS       [SINGER, ALBUM, DATERANGE]
  QUANT      all
  MOD_TYPE   speech
  TIME_ST    00:00:00-00:00:00
  SINGER     [ DMOVE   SpecifyParameter
               TYPE    Singer
               CONT    Queen ]
  DATERANGE  [ DMOVE   SpecifyParameter
               TYPE    DateRange
               ARGS    [DATE1, DATE2]
               DATE1   [ DMOVE   SpecifyParameter
                         TYPE    Date
                         RESTR   [RELATION  FROM]
                         ARGS    [YEAR]
                         YEAR    [ DMOVE   SpecifyParameter
                                   TYPE    Year
                                   CONT    1981 ] ]
               DATE2   [ DMOVE   SpecifyParameter
                         TYPE    Date
                         RESTR   [RELATION  TO]
                         ARGS    [YEAR]
                         YEAR    [ DMOVE   SpecifyParameter
                                   TYPE    Year
                                   CONT    1985 ] ] ]
```

In the previous example, two new main categories can be observed:

QUANT: Short for Quantifier and as it name indicates determines the quantity.

RESTR: Short for Restriction. Its function is to establish the constraints that will determine the search

space. There are different types of restrictions such as those illustrated in the examples (RELATION, ORDER, etc.).

Instead of performing an RDQL query, the Knowledge Manager will now translate this information state into an SQL query:

```
SELECT ["song"] FROM ["MP3DbTable"] WHERE ["singer=Queen"
"Date>1981" "Date<1985"]
```

In our MP3 showcase we assume databases with flat structures and whose fields are contained within the same table.

6.4 Conclusions and Future Work

In this chapter, we described the extensions of the information state in the Delfos system. USE has extended the "Delfos DTAC structure" to allow for multimodal interaction in both complementing and unrelated simultaneous tasks. In order to accomplish this, two new categories have been integrated in the DTAC structure and additional contextual information is used to design the decision algorithm.

Additionally, other extensions have been included in order to enable the DTAC structure to deal with complex queries such as database searches.

As additional functionalities are added and specific scenarios considered, new extensions will be added in order to handle a higher degree of complexity in the interaction with the system

The prototype included in the appendix contains the current implementation of the extended information state within a preliminary version of the Delfos final showcase.

Chapter 7

Conclusions

We have presented the work carried out in Task 3.1, as part of WP3. Task 3.1 contributes to the overall objective of WP3 to support flexible and adaptive system output presentation in multiple modes by investigating and subsequently implementing the extensions that are needed in the information state representations and the corresponding information state updating in the respective showcases.

We started by giving a compact survey of the state of the art in ISU-based dialogue modeling prior to the TALK project. We have seen that although a number of ISU-based systems had been developed which all share the underlying concept of viewing dialogue utterances as update acts on an information state, there are differences in what is represented in the information states, depending on which theory the particular system is based on. Nevertheless, the main advantage of ISU-based dialogue modeling, namely that the abstract representations of dialogue states and update rules allow generic characterization of flexible dialogue strategies, remain and are shared across the systems.

We proposed a range of extensions of information state representations with respect to the systems prior to TALK. The main common denominator of the extensions developed in the various showcases has been to handle multimodal input and output. But in addition, other extensions have been developed that reflect the a range of specific research issues pursued within the different showcases. These extensions are to a large extent complementary.

The information states of the SAMMIE, GODIS and Delfos systems all contain an extended representation of the multimodal discourse context. This involves on the one hand keeping track of the available modalities and their capacity to convey information at any given point, as well as other contextual features helping multimodal fission. In addition, SAMMIE and GODIS also use an extended discourse history representing the previous utterances and/or the corresponding communicated content in a structured way, used in particular for the planning of system output realization as well as for context-sensitive interpretation.

The SAMMIE information state additionally also contains extended task representations reflecting the collaborative problem solving state, in order to support the modeling of planning and execution status and the decision-making process throughout a dialogue, for the purpose of providing collaborative dialogue behavior.

The extensions to the IS made by UEDIN for the UEDIN/UCAM system concern on the one hand a range of features to represent the dialogue context for the purpose of reinforcement learning, namely, the IS fields for task state, dialogue history, and reward, and on the other hand extensions for fragmentary generating fragmentary clarifications, namely the IS field for word-based confidence scores.

To summarize, we have developed a range of information state extensions, which on the one hand allow us to handle multimodal interactions in our ISU-based dialogue systems, and on the other hand support various aspects of advanced multimodal presentation that we pursue in our research. The advantage of the ISU-base approach is that the use of structured Information States as a central repository of contextual information allows straightforward implementation of flexible dialogue systems which can access and modify information in the Information State in different sequences and by varying means.

Appendix A

Prototypes

A.1 The SAMMIE In-Car Dialogue System

DFKI and USAAR currently develop and implement the final showcase system of the DFKI/USAAR in-car multimodal dialog system. On the attached CD/DVD you find an intermediate snapshot of the entire system going beyond the baseline system, i.e., some of the planned new features for the final showcase system are already partly realized in this system. The attached system is sustainably based on the DFKI/USAAR baseline MP3 in-car dialog system SAMMIE of mid 2005.

In the following paragraphs, we give a short description of the attached intermediate system. We first characterize the main features of the DFKI/USAAR baseline system and then describe the already realized new features of the final showcase system.

A.1.1 Characteristics of the DFKI/USAAR baseline system

In the DFKI/USAAR baseline system the user, i.e., the driver, can interact multimodally with an in-car MP3 player, which supports the usual functionalities, including, e.g., song playback and playlist creation and modification.

All these features can also be controlled by speech only (monomodal), but preferably (where possible and reasonable) by speech together with the ErgoCommander controller (multimodal). Due to our focus on multimodal/speech-driven communication, the monomodal usage of the ErgoCommander is restricted to the command and control functionality and also simple browsing of displayed sets/lists.

System output is multimodal in almost all situations, including a spoken message, information on the display such as text, lists or tables, and the player status and controls. Additional system output that simplifies the controlling of the interaction is also multimodal, i.e., there is acoustic and visual feedback about microphone (and thus system) status. For further details we refer to D5.2 (Baseline System for In-Car Showcase). In the following, we just give a short recapitulation of the major processing steps and components of the baseline system.

Input modalities are speech and GUI input. Speech input is recognized by the Nuance speech recognizer and analyzed using Nuance's natural language understanding system. GUI input (e.g., in the form of a selection of a table row) comes from the ErgoCommander in-car device (described below) and forwarded to the Table Presenter. The table presenter converts the ErgoCommander event (click) into a domain object

(e.g., an album) and sends this to the Interpretation Manager. The Interpretation Manager performs the interpretation taking context and multimodal fusion into account. It computes a set of communicative intentions (based on a theory of agent-based dialogue [BA05]) which describe the user's turn (also see deliverables D2.1: Integration of Ontologies and the ISU Approach and D3.1: Extended Information State Modeling). This set of communicative intentions is then sent to the dialogue manager, which is a baseline instantiation of the collaborative problem-solving theory of dialogue described in [BA05]. Using update rules, the dialogue manager integrates user communicative intentions into the information state (IS), and then uses the current IS to drive its behavior: including making queries to the FreeDB database as well as controlling the MP3 Player (both of which are encapsulated by the MP3 Shield, whose task it is to wrap these two components with a simple API). The Dialogue Manager also formulates its own communicative intentions (again using the model in [BA05]) and sends these to the Turn Planner. The Turn Planner then performs multimodal fission, deciding which content to realize graphically (both tabular and written language) and which to speak. Linguistic realizations (i.e., spoken utterances and written sentences) are computed by the Linguistic Planner, currently using template-based generation. The Turn Planner itself packages graphical output. The entire output request is then sent to the Output Manager, which times and coordinates graphical and spoken output, which are realized using the Table Presenter and Mary, respectively.

The first version of the DFKI/USAAR baseline system was delivered on June 30th, 2005. Since then, a number of updates are following to address issues that are necessary for the evaluation of the baseline system that was conducted in October and November of 2005.

A.1.2 New features of the intermediate system version on CD/DVD

The current DFKI/USAAR system on the CD/DVD is an intermediate version of the final showcase system that will be finished in mid 2006. Compared to the baseline system, some of the planned new features are already realized in this system. Here is a short list of them:

- dialog manager core engine switched from JTFS-DIPPER to PATE
- first basic version of the new domain (restaurant information for a city including appropriate maps)
- bilinguality (first English recognizer grammar for MP3 domain added)
- basic user attention realizer integrated
- first complete turn in the new domain implemented and running

A.1.3 Installation and Running

The system comes as a tarball with all internal components. However, some of these components rely on external software. Here is a short list:

- proper installation of Nuance speech recognizer 8.0
- java1.5.3
- proper installation of OAA 2.3.0 (software comes with the system)

For further details we refer to the file README.txt on toplevel of the system distribution.

A.2 GoDiS

The code distributed with this deliverable contains an alpha version of TrindiKit 4, also available from SourceForge.[1] It also contains GoDiS and a number of different GoDiS applications. A readme file is included that shows how to run the system in text mode. The library `godis-apps/` contains `domain-tram/`, where files relevant to the tram domain discussed here can be found. Notably the following files are of interest:

(38)
- `start-tram-text-t4.pl` is the start file for the tram application. It contains the definition of the extended information state, the resource and module interface variables, the setting of default properties, as well as run commands for predefined modes. The application is started by consulting the file, and then typing one of the following at the prolog prompt:

  ```
  run.
  rundriving.
  runtelephone.
  runmeeting.
  runathome.
  ```

  Typing `verb.` *before* any of the run commands enables the inspection of the information state and the application of the update rules during the dialogue, whereas `quiet.` shows only the dialogue. The verb(ose) mode is the default mode.
- `infoamount.pl` contains the definition of the type Infoamount and its values.
- `modeprops.pl` contains the default properties of the different predefined modes.
- `domain_tram.pl` in `domain-tram/Resources/` gives the domain knowledge for the tram domain, including the application's plans.
- `lexicon_tram_english.pl` in `domain-tram/Resources/` contains the mapping between output moves and system utterances, and between user utterances and input moves.

Another important file, `update_rules.pl`, which contains all the new update rules, in addition to old ones, as well as modifications of old update rules, can be found in `godis/godis-aod/`.

The dialogues (33)-(37) above, and variations of these, can all be used with the system to investigate the extended information state. Saying "Top" to the system at any time during the dialogue clears the information state and makes the system start from the beginning.

A.3 The UEDIN/UCAM In-Car Dialogue System

The software demonstrating the UEDIN/UCAM information state consists of the following:

- DIPPER dialogue manager (information state, update rules, resources)
- ATK speech recognizer and Language Models
- Dialogue policy learner agent

[1] http://sourceforge.net/projects/trindikit/

- Festival speech synthesizer agent
- Map agent and map for the SACTI in-car domain
- Database agent and MySQL database for the SACTI in-car domain.

Additional software required to run the system, but not produced by the TALK project (e.g. Java, OAA, Prolog) can be downloaded and installed via the internet. More details about the system can be found in [LGS05].

A.4 DELFOS

The software demosntrating the Delfos information state is a version of the showcase for the in-home scenario, and more specifically for wheel-chair-bound users. The current version of system includes the Multimodal DTAC extensions of Delfos. Once lexicons and grammars are configured, the user may control the devices in speech-only or mixed modes. The new extended DTAC structures can be obtained activating the ”trace” option. The new OWL ontology has not yet been included in this version and the Multimodal Presentation Module is still under development. The prototype includes a README file with precise instructions to configure and run the prototype.

Bibliography

[ABWF+98] Jan Alexandersson, Bianka Buschbeck-Wolf, Tsutomu Fujinami, Michael Kipp, Stephan Kock, Elisabeth Maier, Norbert Reithinger, Birte Schmitz, and Melanie Siegel. Dialogue acts in VERBMOBIL-2 second edition. Verbmobil Report 226, DFKI Saarbrücken, Universität Stuttgart, TU Berlin, Universität des Saarlandes, July 1998.

[AC97] James Allen and Mark Core. Draft of DAMSL: Dialog act markup in several layers. Available at http://www.cs.rochester.edu/research/cisd/resources/damsl/, October 1997.

[AFS01] James Allen, George Ferguson, and Amanda Stent. An architecture for more realistic conversational systems. In *Proceedings of Intelligent User Interfaces 2001 (IUI-01)*, pages 1–8, Santa Fe, NM, January 2001.

[BA05] Nate Blaylock and James Allen. A collaborative problem-solving model of dialogue. In Laila Dybkjær and Wolfgang Minker, editors, *Proceedings of the 6th SIGdial Workshop on Discourse and Dialogue*, pages 200–211, Lisbon, September 2–3 2005.

[BAF02] Nate Blaylock, James Allen, and George Ferguson. Synchronization in an asynchronous agent-based architecture for dialogue systems. In *Proceedings of the 3rd SIGdial Workshop on Discourse and Dialog*, Philadelphia, July 2002.

[BBG+05] Tilman Becker, Nate Blaylock, Ciprian Gerstenberger, Michael Pitz, Peter Poller, and Jan Schehl. Baseline system for in-car showcase. Deliverable 5.2, TALK Project, September 2005.

[BDG+03] Carl Burke, Christy Doran, Abigail Gertner, Andy Gregorowicz, Lisa Harper, Joel Korb, and Dan Loehr. Dialogue complexity with portability? research directions for the information state approach. In *Proceedings of the HLT-NAACL 2003 Workshop on Research Directions in Dialogue Processing, May 2003, Edmonton, Canada*. 2003.

[BG00] Johan Bos and Malte Gabsdil. First-order inference and the interpretation of questions and answers. In Massimo Poesio and David Traum, editors, *Proceedings of Goetalog 2000. Fourth Workshop on the Semantics and Pragmatics of Dialogue. Gothenburg Papers in Computational Linguistics 00-5*, pages 43–50. 2000.

[BKLO03a] Johan Bos, Ewan Klein, Oliver Lemon, and Tetsushi Oka. Dipper: Description and formalisation of an information-state update dialogue system architecture. In *Proceedings of the 4th SIGdial Workshop on Discourse and Dialogue*, Sapporo, Japan, 2003.

[BKLO03b] 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.

[BKO03] Johan Bos, Ewan Klein, and Tetsushi Oka. Meaningful conversation with a mobile robot. In *Proceedings of the EACL*, pages 71–74, 2003.

[Bla05] Nathan J. Blaylock. *Towards Tractable Agent-based Dialogue*. PhD thesis, University of Rochester, Dept. of Computer Science, August 2005. Also available as University of Rochester Department of Computer Science Technical Report 880.

[BR03] Dan Bohus and Alexander I. Rudnicky. RavenClaw: Dialog management using hierarchical task decomposition and an expectation agenda. In *Proceedings of Eurospeech-2003*, Geneva, Switzerland, 2003.

[Car90] Sandra Carberry. *Plan Recognition in Natural Language Dialogue*. ACL-MIT Press Series on Natural Language Processing. MIT Press, 1990.

[CCC00] Jennifer Chu-Carroll and Sandra Carberry. Conflict resolution in collaborative planning dialogues. *International Journal of Human-Computer Studies*, 53(6):969–1015, 2000.

[CL90] Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.

[Cla96] Herbert H. Clark. *Using Language*. Cambridge University Press, 1996.

[CLNO91] Philip R. Cohen, Hector J. Levesque, José H. T. Nunes, and Sharon L. Oviatt. Task-oriented dialogue as a consequence of joint activity. In Hozumi Tanaka, editor, *Artificial Intelligence in the Pacific Rim*, pages 203–208. IOS Press, Amsterdam, 1991.

[DJTM97] Barbara Di Eugenio, Pamela W. Jordan, Richmond H. Thomason, and Johanna D. Moore. Reconstructed intentions in collaborative problem solving dialogues. In *Working Notes of AAAI Fall Symposium on Communicative Action in Humans and Machines*, Cambridge, Massachusetts, November 1997.

[DJTM00] Barbara Di Eugenio, Pamela W. Jordan, Richmond H. Thomason, and Johanna D. Moore. The agreement process: An empirical investigation of human–human computer-mediated collaborative dialogs. *International Journal of Human-Computer Studies*, 53:1017–1076, 2000.

[Eri05] Stina Ericsson. *Information Enriched Constituents in Dialogue*. PhD thesis, Göteborg University, 2005.

[FB03] Gerd Fliedner and Daniel Bobbert. A framework for informations-state based dialogue (demonstration). In Ivana Kruijff-Korbayovaá and Claudia Kosny, editors, *Proceedings of Diabruck - 7th Workshop on the Semantics and Pragmatics of DIalogue (SEMDIAL), Saarbrücken, September 4–6 2003*, pages 179–180, 2003.

[Gat92] Erann Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI)*, 1992.

[GHL05] Kallirroi Georgila, James Henderson, and Oliver Lemon. Learning User Simulations for Information State Update Dialogue Systems. In *Interspeech/Eurospeech: the 9th biennial conference of the International Speech Communication Association*, 2005.

[Gin96a] Jonathan Ginzburg. Interrogatives: Questions, facts and dialogue. In *The Handbook of Contemporary Semantic Theory*. Blackwell, Oxford, 1996.

[Gin96b] Jonathan Ginzburg. Interrogatives: Questions, facts and dialogue. In Shalom Lappin, editor, *The Handbook of Contemporary Semantic Theory*, pages 385–422. Blackwell, Oxford, 1996.

[GLH05] Kallirroi Georgila, Oliver Lemon, and James Henderson. Automatic annotation of COMMUNICATOR dialogue data for learning dialogue strategies and user simulations. In *Ninth Workshop on the Semantics and Pragmatics of Dialogue (SEMDIAL: DIALOR)*, 2005.

[Gri69] H. Paul Grice. Utterer's meaning and intention. *Philosophical Review*, 78(2):147–177, 1969.

[GS86] Barbara Grosz and Candace Sidner. Attention, intention, and the structure of discourse. *Computational Linguistics*, 12(3):175–204, 1986.

[GS90] Barbara J. Grosz and Candace L. Sidner. Plans for discourse. In P. R. Cohen, J. Morgan, and M. Pollack, editors, *Intentions in Communication*, pages 417–444. MIT Press, Cambridge, MA, 1990.

[HLG05] James Henderson, Oliver Lemon, and Kallirroi Georgila. Hybrid Reinforcement/Supervised Learning for Dialogue Policies from COMMUNICATOR data. In *IJCAI workshop on Knowledge and Reasoning in Practical Dialogue Systems*,, 2005.

[Kem04] Benjamin Kempe. PATE a production rule system based on activation and typed feature structure elements. Bachelor Thesis, Saarland University, August 2004.

[LA90] Diane J. Litman and James F. Allen. Discourse processing and commonsense plans. In P. R. Cohen, J. Morgan, and M. Pollack, editors, *Intentions in Communication*, pages 365–388. MIT Press, Cambridge, MA, 1990.

[Lar02] Staffan Larsson. *Issue-based Dialogue Management*. PhD thesis, Göteborg University, 2002.

[LBGK02] Staffan Larsson, Alexander Berman, Leif Grönqvist, and Frederik Kronlid. Trindikit 3.0 manual. Technical Report Deliverable D6.4 - Manual, Trindi, 2002.

[LBGP01] Oliver Lemon, Anne Bracy, Alexander Gruenstein, and Stanley Peters. The WITAS multimodal dialogue system i. In *Proceedings of EuroSpeech 2001*, Aalborg, Denmark, 2001.

[Lew98] Ian Lewin. The autoroute dialogue demonstrator. Technical Report CRC-073, SRI Cambridge Computer Science Research Centre, 1998.

[LGH+05] Oliver Lemon, Kallirroi Georgila, James Henderson, Malte Gabsdil, Ivan Meza-Ruiz, and Steve Young. D4.1: Integration of Learning and Adaptivity with the ISU approach. Technical report, TALK Project, 2005.

[LGP02] Oliver Lemon, Alexander Gruenstein, and Stanley Peters. Collaborative activities and multi-tasking in dialogue systems: Towards natural language with robots. *Traitement Automatique des Langues (TAL)*, 43(2):131–154, 2002.

[LGS05] Oliver Lemon, Kallirroi Georgila, and Matthew Stuttle. D4.2: Showcase exhibiting Reinforcement Learning for dialogue strategies in the in-car domain. Technical report, TALK Project, 2005.

[LT00] 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.

[Lup91] Susann LuperFoy. *Discourse Pegs: A Computational Analyses of Context Dependent Referring Expressions*. PhD thesis, University of Texas at Austin, December 1991.

[MAB+05] David Milward, Gabriel Amores, Tilman Becker, Nate Blaylock, Malte Gabsdil, Staffan Larsson, Oliver Lemon, Pilar Manchón, Guillermo Pérez, and Jan Schehl. Integration of ontological knowledge with the ISU approach. Deliverable 2.1, TALK Project, September 2005.

[MPT00] 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 Linguistic*, May 2000.

[MT87] William C. Mann and Sandra A. Thompson. Rhetorical structure theory: A theory of text organization. In L. Polanyi, editor, *The Structure of Discourse*. Ablex Publishing Corporation, 1987.

[PAB03] Norbert Pfleger, Jan Alexandersson, and Tilman Becker. A robust and generic discourse model for multimodal dialogue. In *Proceedings of the 3rd Workshop on Knowledge and Reasoning in Practical Dialogue Systems*, Acapulco, 2003.

[Pfl04] Norbert Pfleger. Context based multimodal fusion. In *Sixth International Conference on Multimodal Interfaces (ICMI'04)*, State College, Pennsylvania, 2004.

[SP00] Stephanie Seneff and Joseph Polifroni. Dialogue management in the Mercury flight reservation system. In *Proceedings of ANLP/NAACL-2000 Workshop on Conversational Systems*, Seattle, Washington, May 2000.

[TBC+99] David Traum, Johan Bos, Robin Cooper, Staffan Larsson, Ian Lewin, Colin Matheson, and Massimo Poesio. A model of dialogue moves and information state revision. Deliverable D2.1, TRINDI, 1999.

[TH92] David R. Traum and Elizabeth A. Hinkelman. Conversation acts in task-oriented spoken dialogue. *Computational Intelligence*, 8(3):575–599, 1992. Also available as University of Rochester Department of Computer Science Technical Report 425.

[Tra94] David R. Traum. A computational theory of grounding in natural language conversation. Technical Report 545, University of Rochester, Department of Computer Science, December 1994. PhD Thesis.

[Tra00] David R. Traum. 20 questions for dialogue act taxonomies. *Journal of Semantics*, 17(1):7–30, 2000.

[Val92] Enric Vallduví. *The Informational Component*. Garland, 1992.

[Val01] Enric Vallduví. Fragments in information packaging. Talk at the ESSLLI'01 workshop on Information Structure, Discourse Structure and Discourse Semantics, 2001.

[WKL00] Marilyn A. Walker, Candace A. Kamm, and Diane J. Litman. Towards Developing General Models of Usability with PARADISE. *Natural Language Engineering*, 6(3), 2000.

[ZMC+03] Claus Zinn, Johanna Moore, Mark Core, Sebastian Varges, and Kaska Porayska-Pomsta. The BE&E tutorial learning environment (BEETLE). In *Proceedings of the Seventh Workshop on the Semantics and Pragmatics of Dialogue (DiaBruck 2003)*, Saarbrücken, Germany, September 2003.