Master's Thesis

An Agent-Based Platform for Dialogue Management

Mark Buckley

December 2005

Prepared under the supervision of
Dr. Christoph Benzmüller

Hiermit versichere ich an Eides statt, dass ich diese Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Saarbrücken, 19. Dezember 2005

Acknowledgements

Abstract

In this thesis we investigate the application of agent-based techniques to the field of dialogue management. We develop a platform upon which a dialogue manager can be built which supports the information state update approach to dialogue management. It will use agent technology and a hierarchical design to achieve flexibility and concurrency in the integration and interleaving of modules such as linguistic processing and domain reasoning in a dialogue system. The research is done in the framework of the DIALOG project, which investigates flexible natural language tutorial dialogue on mathematical proofs. There are two main contributions of this thesis. The first is the design and implementation of a dialogue manager for the demonstrator of the DIALOG system. The second is the Agent-based Dialogue Management Platform, ADMP. We give a formalisation of ADMP and show how it can be used to implement a dialogue manager for the DIALOG project.

Contents

List of Figures

List of Tables

1

Introduction

Natural language is becoming increasingly important as a way to interact with machines. As computer systems become more and more integrated into everyday tasks, the significance of natural language dialogue as an interface is also increasing. This is facilitated by continuing improvements in the sophistication and effectiveness of speech technology and language processing, as well as developments in multi-modal interfaces. Using a spoken interface to a computer system can have great benefits — it can increase speed, and allows the user to work "hands-free" in comparison to a textual interface, and is a more natural way to use a system for an untrained user.

There is a wide range of scenarios in which dialogue systems are moving from being research prototypes to becoming real-world applications. Information-seeking systems use dialogue as a front-end for example to timetabling or financial applications. Collaborative planning systems support the user in solving a task together with the system, for instance repairing a machine, during which the user can discuss plans or solutions with the system in natural language.

A more complex application is for instance an e-learning system. Here the dialogue system must form the interface to many subsystems such as a source of domain knowledge, a user model, or a source of pedagogical reasoning. Such a natural language tutorial domain will form the context of this research.

This thesis is concerned with a subdiscipline of dialogue systems known as dialogue management, and with the application of agent-based techniques in the field of dialogue management. In this chapter we briefly introduce the theories that are required to develop dialogue management, and then give our motivations and goals for this research as well as an overview of the structure of the thesis.

1.1   Agent-Based Dialogue Management

Dialogue management involves controlling the flow of the dialogue between a system and a user, and orchestrating the system's execution. These functions are usually encapsulated in a part of the system known as the dialogue manager.

The overall goal of the thesis is to investigate the use of agent techniques in dialogue management. Concretely, we will use agent technology to build a platform for dialogue management. A platform in this sense is a framework which can be used to instantiate a dialogue manager.

Dialogue management draws on many different areas of research. In developing our dialogue manager we will introduce two broad areas of research which will form the basis of our research: dialogue modelling and agent technology.

Dialogue Modelling

Managing dialogue depends a theory of dialogue modelling. Dialogue modelling is the representation of those aspects of the dialogue which influence its state and its flow. A dialogue model can contain, for example, a representation of what objects have been addressed in the dialogue, what utterances have been performed, or a representation of the internal state of the dialogue participants.

A theory of dialogue modelling depends on a notion of discourse. Discourse is a general term which captures any kind of linguistic interaction. Theories of discourse describe its structure, meaning, and how its form is licenced. Conversation analysis is a field of research which attempts to describe dialogue and its characteristics. Formal theories of conversation analysis are used in the representation of a dialogue model, and thus in dialogue management.

Agent Technology

A very general notion of an agent is "something that perceives and acts in an environment"[1]. In this thesis however, we will concern ourselves with a more restricted view, that of a software agent. This is a software process which runs independently of other software agents. Software agents can be used to collaboratively solve problems or carry out computations in a distributed manner. We will use software agents as the basis of our dialogue management framework.

1.2   Overview of the Thesis

In this thesis we investigate the application of agent-based techniques in dialogue management. Our motivation for this proposal is twofold: on the one hand we are motivated by

---

[1][80], page 49.

our experiences in developing a previous dialogue manager for the Dialog project, and on the other we are motivated by the application of agent technology in the Ω-Ants system.

This research has been done in the framework of the Dialog project, an interdisciplinary project with the goal of investigating the issues associated with flexible natural language dialogue in a mathematical tutorial environment. Our work is motivated by our experiences with the Dialog demonstrator, a system implemented to show the extent of the progress of the project. A dialogue manager was specially built for this system. Based on this we identified some features which are desirable for a dialogue manager in the Dialog scenario. An example is support for the information state update approach, a theory of dialogue management.

Our second motivation is the Ω-Ants project. Ω-Ants is a suggestion mechanism for interactive and automated theorem proving. It uses an agent-based architecture with a hierarchical structure to achieve concurrency, maximise resource efficiency, and easily integrate external systems. We argue that by borrowing from the agent-based hierarchical design of Ω-Ants, we can build a dialogue manager that shares these benefits.

Contributions of the Thesis

Like the motivations of the research, the contribution of this thesis is also twofold. The first is the design and implementation of the dialogue manager for the Dialog demonstrator, and is presented in Chapter 5. The second is the Agent-based Dialogue Management Platform (Admp). This is the main contribution of the thesis, and is presented in Chapter 7. Admp is a reusable platform for dialogue management which can be deployed in tutorial scenarios, and which will use the information state update approach. It will therefore be suitable for employment in the Dialog scenario. It will apply in its design the agent-based technology and the hierarchical design used in Ω-Ants.

Structure of the Thesis

This thesis is divided into two parts. Part I expounds the theories of dialogue management and introduces the reader to the Dialog project. Part II presents the agent technologies which will be used in Admp, and gives an formal account and discussion of Admp itself.

Part I begins in Chapter 2 by introducing the reader to the basic notions of discourse, dialogue and dialogue modelling. We then continue in Chapter 3 with a treatment of dialogue systems and their use of dialogue management theories, including the information state update approach. Chapter 4 presents the Dialog project itself, followed in Chapter 5 by an account of the dialogue manager for the Dialog demonstrator.

Part II deals with agent-based dialogue management. It begins with a description of Ω-Ants in Chapter 6. Here we show the agent technologies which are used in Ω-Ants and which we will use in Admp. Chapter 7, the main contribution of the thesis, presents Admp in detail. Here we give its design and a formalisation of the system, and show how a dialogue manager can be built using it. We follow this in Chapter 8 with an example system and an evaluation. Chapter 9 summarises the thesis and gives an outlook.

Part I

Dialogue Management

2

Dialogue Modelling

2.1 Introduction

This chapter is an introduction to the concepts of discourse and dialogue, and forms part of the theoretical basis of the rest of the thesis. We first give a description of discourse, its associated phenomena, and discourse structure. We then concentrate on dialogue itself as a subtype of discourse. We treat the concepts of speech acts, beliefs, the common ground of the dialogue, and implicature. Finally we present some accounts which model the dialogue context. The notions of dialogue modelling presented here will form the background of our discussion of dialogue management in the next chapter.

2.2 Discourse

The term discourse captures, in a very general sense, any kind of linguistic interaction. Clark [24] embeds discourse in the notion of *joint activity*, arguing that discourse is a joint activity in which language plays a prominent role. Successfully completing a joint activity requires communication, and as such a discourse can be seen as a medium for communicating using language.

At a more concrete level we can define a discourse as a group of sentences or utterances which stand in some relation to one another. Examples include text, speeches, and both written and spoken dialogue. Exactly what the relations between utterances are is constrained by the content of the utterances, the intentions of the speaker or writer, and the background context of the dialogue.

In this section we will present some of the properties of discourse and some theories which describe them. We then briefly mention some types of discourse.

2.2.1   Properties of Discourse

In this section we introduce a number of properties of discourse. We begin with coherence, which is a property of a discourse which "makes sense" as a sequence of sentences. To analyse a discourse it is necessary to consider its inner structure, that is, how the meaning or content of the sentences relate to each other. We present some theories of discourse structure which describe these hierarchical relationships. Finally we describe cohesion, which is a measure of how well a discourse "hangs together", and mention some features which contribute to it.

Coherence

In order to be understandable a discourse should "make sense" as a sequence of sentences. This property is known as *coherence*, because the discourse should describe a coherent state of affairs or a coherent sequence of events. For example, the discourse[1] in (1a) is coherent because the reader immediately perceives the causal link between the two sentences.

(1)   a. John hid Bill's car keys. He was drunk.

       b. ?? John hid Bill's car keys. He likes spinach.

The discourse in (1b) however would typically be considered incoherent, because this causal relation is not apparent. We see that coherence is heavily dependent on the current context, because although it is common knowledge that being drunk is a reason not to be allowed drive, it is not apparent what the link is between liking spinach and not being allowed drive.

   As a representation of the causal relation between sentences Hobbs [49] proposes *coherence relations* which can hold between sentences in a discourse. Each relation is accompanied by conditions that must be satisfied for the relation to hold. For instance, the relation EXPLANATION holds between consecutive sentences S_0 and S_1 if the state or event asserted by S_1 causes or could cause the state or event asserted by S_0. This is the relation that licences the coherence of (1a). To establish the coherence of a discourse, it suffices then to show a coherence relation for each pair of consecutive sentences.

Discourse Connectives   Certain words or phrases can serve as a signal for what coherence relation holds between two sentences or clauses. For instance, the word "but" indicates a contrasting relation. These are adverbial cue phrases or *discourse connectives*. The mapping of discourse connectives to coherence relations is however not always unique, as shown by "and", which can indicate a number of coherence relations, such as PARALLEL or RESULT. A single coherence relation such as CAUSE can also be indicated by more than one connective, in this case both "because" and "seeing as".

   Cohen [26] distinguishes two functions of discourse connectives. Firstly, they enable faster recognition of the coherence relation which holds between the clauses, and secondly,

---

[1]from [52]

they allow the recognition of relations which would otherwise be uninferable. In this way discourse connectives contribute to the overall coherence of a discourse.

Discourse Structure

Up to now we have only considered relations which hold between pairs of sentences, but in fact we can consider relations at a more abstract level. For instance, a sentence may stand in a coherence relationship to a sentence which does not directly precede it in the discourse. Also, a sentence can be related not just to a single sentence, but to a group of sentences. We refer to such a group of locally coherent sentences as a *discourse segment*. This means we can analyse discourse in terms of a hierarchical structure in a similar way to the syntactic structure of a sentence. In this section we present some of the established theories of discourse structure.

Rhetorical Structure Theory (RST)   Mann and Thompson [61] propose a theory of discourse structure which relates discourse segments. It describes the structure of the text by relating its segments to one another using *rhetorical relations*. The theory defines a set of 23 relations, some of which are subtypes of others. An example is ELABORATION, which has as subtypes whole-part, set-member, among others.

Since many discourse relations are asymmetric, most RST relations differentiate between a central and peripheral segment known as the nucleus and satellite respectively. For instance the relation RESULT has as its nucleus the segment which results from the one in the satellite segment. Others, such as CONTRAST are multi-nuclear. Relations are used to build a tree structure whose nodes correspond to discourse segments and whose root node represents the discourse segment which contains the whole discourse. RST was conceived as a descriptive theory, but can also be used as a descriptive tool for discourse planning in natural language generation.

Grosz and Sidner   Grosz and Sidner [45] propose a theory of discourse structure based on three constituents: the linguistic structure, the intentional structure and the attentional state. The linguistic structure is the structure of the utterances in the discourse, like the words in a sentence. It is made up of discourse segments which have a function in the overall discourse. Nonconsecutive utterances can be in the same discourse segment; equally consecutive utterances may be in different discourse segments. This subdivision into segments has been analysed in many types of discourse [43].

The second constituent is the intentional structure, which encodes the purpose that underlies a discourse. The intention of the discourse is known as the discourse purpose. This is closely linked to the linguistic structure, since each discourse segment has a discourse segment purpose. This is the intention of that segment of the discourse which contributes to the satisfaction of the discourse as a whole.

The third constituent is the attentional state, which models the salience of objects, propositions, relations, etc. The salience of an object refers to the degree to which it is accessible by a hearer or speaker in his mental model of the discourse. The attentional

state is modelled by a set of focus spaces, each associated with a discourse segment. The set of focus spaces is modelled as a stack, reflecting the accessibility of salient entities.

Centring theory   Centring theory [44] fits within the theory of discourse structure developed by Grosz and Sidner introduced above, and models the local component of the attentional state. It accounts for the choice of referring expressions within a discourse segment by highlighting entities which are *centred* in the utterance. The centres of an utterance are the entities which serve to link the utterance to the others in its discourse segment. The backward looking centre is the entity being focused when the utterance is interpreted. The forward looking centres are those which can become the backward looking centre for the next utterance. Pronouns can be resolved using centring by giving a partial order on the forward looking centres of the previous utterance, for instance based on grammatical role. The highest ranked centre is then chosen based on constraints such as agreement.

Discourse Representation Theory (DRT) and SDRT   In DRT [53] the content of a discourse is represented by a *discourse representation structure* (DRS). A DRS is a recursive structure consisting of a set of entities salient in the discourse, namely the discourse referents, and a set of conditions on those entities imposed by the discourse. The DRS for some discourse segment is constructed out of the DRSs of the constituent segments. A discourse can also be interpreted in an incremental way by adding the content of a new discourse segment to the existing DRS.

   DRT does not however account for discourse structure. To do this, Asher [7] proposes an extension to DRT called *segmented DRT* (SDRT). The theory adds an extra layer of structure called a segmented DRS (SDRS). SDRSs contain DRSs and conditions on those DRSs. The conditions indicate the discourse relations which hold between the DRSs, and are divided into rhetorical relations such as EXPLANATION or NARRATION and coherence relations such as CAUSE. Like in RST, an SDRT description of a discourse is a tree-like structure, and the relations which licence the tree impose right-frontier restrictions on where new information can be attached in an incremental analysis.

Cohesive Devices

Cohesive devices [46] are phenomena in a discourse which serve to tie its parts together. A discourse is cohesive if its ideas are clearly linked and easy to follow, and such a discourse puts a low cognitive strain on a reader or hearer.

Coreference   A very frequently occurring phenomenon in discourse is *coreference*. This is the process by which speakers refer to entities which are the topic of the discourse. The reference is made by using a referring expression, for instance a name or a pronoun. Two referring expressions which denote the same entity are said to corefer, such as in example (2):

(2)   John bought *a basketball*. He pumped up *the ball* and showed *it* to Mike.

Here the definite noun phrase *the ball* and the pronoun *it* refer to the object introduced by *a basketball*, which is known as the antecedent of the referring expressions. This type of reference is anaphora, and the referring expressions are anaphors.

Coreference is constrained by the discourse context, which is a description of the state of the discourse and the entities which are salient in it. A speaker can use an anaphor to refer to an entity when he believes that the entity is salient for the hearer. If a hearer or reader cannot resolve the anaphors in a discourse, the discourse becomes incoherent. Coreference is also important for both natural language understanding and generation, and computational approaches to resolving anaphors are often based on theories of discourse structure mentioned in the previous section.

Lexical Devices   Choice of words can greatly influence the cohesion of a discourse. Lexical repetition, where a key word or phrase is repeated in sequence of sentences, can help to emphasise the core idea of the discourse. Similarly the use of synonyms keeps the focus of the discourse on a single concept. At a structural level discourse connectives, as introduced in the previous section, contribute to cohesion by making the relationships between clauses more explicit.

2.2.2   Types of Discourse

At the most general level discourses can be classified along two main axes[2]: modality and number of participants. The modality of the discourse is the medium through which language is communicated, and is usually text or speech. There are also cases of multi-modal discourse, for instance in a domain where graphics are used in addition to language.

Monologue and multi-party discourse differ in the number of participants in a discourse. A monologue is a discourse with a single participant, and is realised as a text or non-conversational speech, depending on modality. Monologues are well suited to analysis in terms of the theories presented in this section, since they are typically amenable to segmentation due to their well-defined structure.

Multi-party discourse involves more than one participant, and can be equated with conversation. A discourse consisting of two participants is a dialogue. A discourse with more than two participants can be analysed as a set of discourses between pairs of participants, but such discourses introduce additional issues such as speaker and addressee identification.

In the following we will concentrate on dialogues. The theories we have seen so far in relation to discourse apply in general to dialogue, but additional frameworks must be introduced to handle dialogue-specific phenomena such as turn-taking and grounding. The field of research concerned with dialogue phenomena is conversation analysis, and is the topic of the next section.

---

[2]This does not take account of other classifications such as genre; this is treated in Chapter 3.

2.3   Conversation Analysis

We now focus our attention on dialogue. A dialogue is a spoken, typed or written interaction in natural language between two dialogue participants. Dialogue is a type of discourse, which means that the properties presented in the previous section, such as coherence, cohesion, and accounts of discourse structure, are also applicable. Dialogue however exhibits a number of features which are not found in other forms of discourse. In this section we outline some of these.

2.3.1   Turn-taking

To facilitate useful communication dialogue participants must share the floor. The role of speaker and hearer alternates between participants as each takes and releases control of the floor in order to speak. This characteristic of dialogue is known as *turn-taking*.

Typical conversation contains around 5% overlapping utterances, and silent phases last only a few tenths of a second [57]. Also, turn-taking is not governed by some overall structure of the interaction. This means that speakers must be able to figure out when the turn changes and to whom at a given point in the dialogue. Sacks [81] argues that turn-taking obeys rules which apply at transition-relevance places in the dialogue, in other words points where the turn can change hands. These occur when the speaker offers the turn (for example with a question or by remaining silent) or when the next speaker self-selects by interrupting.

A notion related to turn-taking is that of *adjacency pairs* [57, 82]. These are pairs of adjacent utterances produced by different speakers in which the first speaker offers the turn to the second. They are ordered into a first and a second part and the first part restricts the type of utterance which can occur as its second part. Examples of adjacency pairs are question-answer, greeting-greeting or offer-acceptance. A transition-relevance place is strongly marked if the utterance which appears as a second part of the pair does not conform to the restriction made by the first, or if the first part is met with silence.

2.3.2   Speech Acts

In [9], Austin proposes that in saying something, a speaker is performing an action in the world. The action has an effect in the world which is not necessarily limited to a linguistic function. Austin calls this action a *speech act*. For instance, a performative sentence, like "I baptise you...", carries out the action that the sentence describes. Speech acts however are not restricted to just the performative sentences, and include three kinds of acts:

Locutionary act The act of performing words in sentences that make sense according to grammar and have meaning. This includes the phonetic aspect of speaking.

Illocutionary act The actual act that the speaker wants to perform by uttering the sentence.

Perlocutionary act The effect that an utterance has on the thoughts, feelings or attitudes of the listener. This can be for example the effect of an insulting or surprising utterance.

The term speech act is generally used to describe the illocutionary aspect of an utterance.

Searle [83, 84] improved on Austin's classification of illocutionary acts by modifying the taxonomy. He proposes five major classes, including assertives, directives, commissives, expressives and declarations.

Dialogue Moves As a result of research in dialogue systems the core notion of a speech act has been extended to include for instance aspects of the relationship of the speech act to the rest of the dialogue. A speech act augmented with such dialogue level information is referred to as a dialogue act, or *dialogue move*. A dialogue move uses dimensions to encode the different functions of the utterance, and these summarise the intentions of the speaker. The backward-looking function encodes the relationship of the utterance to the preceding discourse, and the forward-looking direction constrains the future beliefs and actions of the participants, and affects the discourse. The forward-looking function corresponds to the purpose of the utterance, and is close to Austin's speech act. An example is (3), which has as its backward-looking function the reference (*that*) to a previous utterance in the discourse, and has the forward-looking function of imposing an obligation on the hearer to give an explanation.

(3)   Could you explain that please?

One widely used schema for tagging the utterances with dialogue moves is DAMSL (Dialogue Act Markup in Several Layers) [4]. It provides a top level structure for an ontology of dialogue moves, and has as its dimensions forward-looking function, backward-looking function, communicative status (whether the utterance was intelligible and successfully completed) and information level (the semantic content of the utterance). DAMSL provides for a fine-grained description of dialogue moves — for instance the forward-looking dimension contains eight distinct aspects.

The intention is that the DAMSL annotation scheme is refined and extended to account for specific phenomena in a given dialogue genre. We will see an example of this in Section 4.3.2.

2.3.3   Beliefs, the Common Ground and Grounding

For a hearer to correctly understand all of the content of a speech act, knowledge about the state of the world and about the beliefs of the speaker are necessary [24]. The hearer can also base understanding on his/her own beliefs about the world. For a dialogue to function, that is, for successful communication to take place, each dialogue participant should have some concept of what he believes and what he believes his dialogue partner believes at some point in the dialogue. These are their *mutual beliefs*. Both participants

have what they believe to be the common knowledge held by the dialogue participants, known as the *common ground* [90], or the background of the conversation.

The common ground is an important component of the *dialogue context*. This is a representation of the state of the dialogue and its participants. The dialogue context can also be seen as a special case of the discourse context introduced in Section 2.2.1, and includes other dialogue-level information in addition to the common ground.

Grounding   The process by which the common ground is established and maintained during a dialogue is known as *grounding*. In performing grounding, the shared beliefs of the dialogue participants are constantly aligned as new information is added to the dialogue. The term grounding can refer to both the performance of a grounding utterance like "I see..." and the act of assimilating commonly held information.

Some approaches, including Grosz and Sidner's theory above, work under the assumption that assertions are simply added to the common ground, but this is an idealised situation. Non-understanding, and therefore non-grounding, can be signalled by using a token like "Pardon?", meaning that an utterance has not been fully understood, and can therefore not be grounded. Here a speaker should respond by repeating or somehow clarifying his last utterance. However, it is not always clear whether an utterance has been grounded or not.

Clark and Schaefer [25] see grounding as the process of adding information to the common ground by way of contributions. A contribution is accepted by one of five methods, including continued attention, making a relevant next contribution, and acknowledgement of the utterance to be grounded. This is often done by giving an acknowledgement token such as "OK" or "Mm hmmm", also known as backchannelling.

Traum [93] proposes a computational approach in which the content of an utterance is only grounded when the speaker receives explicit feedback, for instance a relevant question. The hearer must ground the utterances that a speaker makes, in other words, confirm that the utterance has been understood and accepted. The theory proposes a set of *grounding acts*, which are the actions performed in producing utterances which contribute to grounding. These include *accept*, *reject* and *repair*. Introducing acts which perform grounding avoids the problem in Clark and Schaefer's contribution theory that acceptances also have to be accepted.

2.3.4   Conversational Implicature

The common ground forms the basis for the interpretation of the meaning of utterances in a dialogue. Grice [42] argues that meaning is actually the combination of saying and implicating, and that the meaning of an utterance in a dialogue goes beyond its literal meaning. This is known as *conversational implicature*. Depending on the dialogue context, the hearer can draw certain inferences from the utterance to conclude what it was the speaker intended. The example

(4)   A: Can you pass the salt?
      B: – *passes salt* –

shows that the intention of the speaker (getting the salt which was out of reach) is inferable given the literal meaning of the utterance (the question of whether B is able to pass the salt) and the context of the dialogue (sitting at a dinner table).

Exactly what inferences can be made are constrained by Grice's maxims. In fact the maxims restrict what speakers can say in order that the correct inferences can be made by the hearer. The maxim of quantity states that the speaker should not be more or less informative than is required. The maxim of quality obliges the speaker not to say what he believes is false. The maxim of relation enforces relevance, and the maxim of manner restricts things like obscurity and ambiguity.

2.3.5   Representations of Dialogue Context

We have seen in the last section that there is a close interaction of dialogue context and speech acts as a dialogue progresses. In order to model this interaction and to facilitate the accurate maintenance of the dialogue context over time, a number of representations have been proposed, which we will look at in this section.

In making an analogy between a baseball game and a dialogue game, Lewis [58] proposes the *conversational scoreboard* as a representation of the current state of the dialogue. The scoreboard is a list of the values of all contextual parameters which describe the dialogue. What can occur in the dialogue is constrained by the values in the conversational scoreboard, and its values evolve in a rule-governed way.

Stalnaker [90] proposes a context which represents the commonly accepted information at a given point. At some time point t there is a set of assumptions which are commonly held at t. When an utterance is made, its descriptive content can be added to the context if it is not inconsistent with the context. However, as Ginzburg [38, 39] argues, this view of context, due to its lack of an inner structure, does not account for the *discursive potential* of the dialogue. In Stalnaker's account new things have as a precondition the totality of what has been hitherto accepted into the common ground, which is not always the speaker's intention.

Ginzburg proposes a notion of context which, in addition to the common ground, explicitly represents what is being discussed in the dialogue at time t. He extends the discourse context with LATEST-MOVE in order to introduce an aspect of locality into the context. LATEST-MOVE stores the syntactic and semantic content of the newest utterance of the dialogue.

However not all utterances relate to the utterance directly preceding them, and for this reason Ginzburg proposes a more general account of what is being discussed in a dialogue, known as *questions under discussion* (QUD). This is a partially ordered set of questions which are currently being discussed, and its maximal element is the current topic of discussion. New questions are added to the top of the QUD as new topics of discussion. A question can be removed from the QUD if information is added to the common ground

which decides the question, or which indicates that no information about the question can be provided.

Dialogue modelling using questions under discussion will be important in Section 3.4.4 where we introduce information state update based dialogue management. In this theory it is used to account for question accommodation, whereby answers which give more information than the question requested can be correctly integrated into the dialogue context.

2.4   Summary

In this chapter we have introduced the main ideas of dialogue modelling. We began with discourse and discourse structure before focusing on dialogue and its properties. We finally outlined some theories of dialogue context modelling. These provide a description of the state of the dialogue which can then be used to constrain or licence actions which move the dialogue forward. For dialogue systems such an explicit model of dialogue context is essential, since the action taken by the system is informed by the dialogue model.

In Chapter 3 we will give an overview of dialogue systems. A central part of any dialogue system is a dialogue manager. Its role is to maintain the dialogue context, and using this, to coordinate the flow of the interaction between the system and the user based on a theory of dialogue. In this way dialogue management depends heavily on dialogue modelling, and we will see some approaches to this in the next chapter.

3

Dialogue Systems

3.1   Introduction

In this chapter we present a review of dialogue systems. We begin in Section 3.2 with a classification of systems according to the types of dialogue that they model. In this classification we consider the subtypes of practical dialogues. Practical dialogues are examples of joint activities as described by Clark. A dialogue system provides a natural language interface for the user and performs some task. In order to achieve this there are a number of functions that must be carried out within a dialogue system, such as natural language processing, domain processing and dialogue management. In Section 3.3 we give a generic architecture of a dialogue system where each of these functions is introduced.

Section 3.4 is an introduction to theories of dialogue management. Dialogue management is the function fulfilled by the dialogue manager, and involves coordinating the interaction between user and system, and maintaining the context of the dialogue. Our treatment of dialogue management builds on the concepts of dialogue modelling described in the last chapter, such as representations for dialogue context. The current chapter also provides the background for our discussion of the DIALOG project in Chapter 4.

3.2   Types of Dialogue Systems

In this section we will give an informal classification of dialogue systems based on the genre of the dialogue that they model. We concentrate on what Allen *et al.* [3] define as *practical dialogues*, that is, interactions in which the dialogue is focused on accomplishing a concrete task. This excludes genres such as casual conversation, in which the unbounded nature of the dialogue makes it less suitable for computational modelling.

Types of dialogue can vary over a number of different features. The initiative in a dialogue refers to which dialogue participant is driving the conversation at some point. The initiative may lie with the system, with the user, or with both, so-called mixed-initiative. In this case both system and user can introduce new topics into the discourse. Dialogue genres also vary with respect to domain. Although the basic principles of dialogue modelling which we introduced in the last chapter are largely domain-independent, domain variations affect for instance the overall dialogue task complexity [3]. This also leads to differences in the degree of natural language processing required by the system.

The concrete realisation of a dialogue system depends to an extent of the type of the dialogue. Simple menu based interfaces can be done over the telephone, whereas some problem solving or e-learning scenarios may require a multi-modal interface using both speech and graphics. We now look at some dialogue genres which are subclasses of practical dialogue.

Information-seeking dialogue   A common application of dialogue systems is as a front-end to a database. Here the dialogue system acts as a natural language interface, and such dialogues are referred to as information-seeking dialogues. The system tries to elicit enough information from the user as is needed to search for the information that the user wants. This means that the initiative in information-seeking dialogues is typically with the system.

The dialogues use a relatively restricted language, so that keyword spotting can often be used to extract the content of the user's utterances. Also, the dialogues follow standard patterns. Examples of dialogue systems which model information-seeking dialogue are ATIS [48], in which a corpus of flight information dialogues has been collected, and the Philips automatic train timetable information system [8].

Negotiation dialogue   The task of negotiation dialogues is that the dialogue participants come to agreement on an issue. Negotiation dialogues differ from many other user/system interactions because in a negotiation both parties will have their own goals and constraints. An example is Verbmobil [99], which models human-human appointment negotiation dialogues. Negotiation is performed at many levels [27], namely at the domain level, the dialogue strategy level, and the meaning level.

Command/control dialogue   In a scenario where the dialogue task is the execution of commands, we speak of natural command languages [6], such as spoken interfaces to VCRs or natural language computer commands. The initiative is fully with the user. The system is also unaware of the user's goal in the interaction, although the range of possibilities is small. Command languages typically have a restricted vocabulary and command dialogues have relatively few states.

A more complex command language dialogue is modelled in the TALK Project [91]. The scenario is controlling an in-car MP3 player, and the interface is multi-modal. The user controls the system at the same time as driving the car, which adds an extra cognitive load.

Problem-solving dialogue   When the user and the system collaborate with the common goal of achieving a complex task, such dialogues are called problem-solving dialogues. The system often models a domain expert who helps the user achieve the task at hand, but there can also be mixed-initiative when the system introduces goals or informs the user of external events. Due to the collaborative nature of this genre, there will often be negotiation subdialogues [16]. Collaborative dialogues also contain many grounding utterances.

The TRAINS project has collected a corpus [47] of problem-solving dialogues in which the user collaborates with a planning assistant to complete a task in a railroad freight system. The follow on project, TRIPS [5], shows how the model can be applied to a number of collaborative domains [2] including kitchen design and disaster relief management.

Tutorial dialogue   The task in tutorial dialogue is that the user, or student, learns concepts or techniques in a given domain. The system sets the user a task or an exercise which should be solved, and then aids the user in finding a solution. The initiative in tutorial dialogue is shared between user and system. The system poses the exercise at hand, but the user is free to raise questions for instance about unknown concepts. Both can initiate clarification subdialogues.

Tutoring should use flexible natural language in order to be effective [70]. However the language can be restricted by concentrating on the domain at hand. Tutorial dialogue systems rely heavily on both domain reasoning and general pedagogical strategies to support the tutorial task. On a wider scale, tutorial dialogue can form part of a broader e-learning application.

There are a number of tutorial systems which use a dialogue interface. The PACT Geometry Tutor [1] models tutoring with knowledge construction dialogues for the physics domain. These allow the student to build up his own knowledge by conversing with the system. AUTOTUTOR [41] also deals with the physics domain, and includes prosodic features and facial expression in the model.

An extensive corpus [29] of tutorial dialogues has been collected by the BE&E project to investigate initiative. Their system, BEETLE [102], aims to tutor basic electronics, and employs multi-turn tutorial strategies. ITSPOKE [60] is a spoken dialogue tutor which builds on Why2-Atlas [97]. Why2-Atlas uses typed dialogue for tutoring in the physics domain. ITSPOKE takes the student state into account, measuring for instance frustration based on prosody. In the mathematical tutoring domain the DIALOG project [72, 73] is investigating flexible approaches to tutoring, domain processing and natural language analysis. We give a full description of the DIALOG project in Chapter 4.

3.3   Components of a Dialogue System

A dialogue system is typically broken down into modules or subsystems that provide the functionality necessary for natural language dialogue. This functionality involves at least natural language understanding, natural language generation, and some interface to external knowledge sources. A control module organises the flow of the dialogue and facilitates
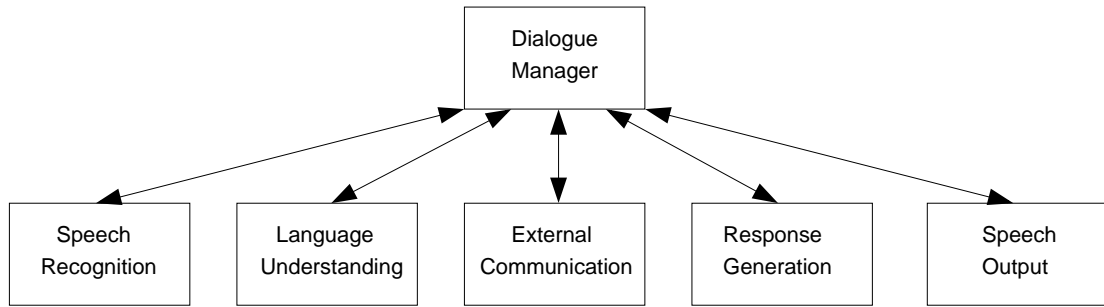
Figure 3.1: General architecture of a spoken dialogue system (from [66], page 113).

communication between modules in such a way that they can interleave correctly. In this section we present the role played by each of these modules.

McTear [66] proposes a general architecture for spoken dialogue systems, shown in Figure 3.1, which includes additional modules that provide speech input and output. Each module is linked to a dialogue manager which controls the system. Further optional parts of a dialogue system include support for other media such as graphics or pointing interfaces.

3.3.1 Natural Language Understanding

Natural language understanding (NLU) is concerned with the analysis of the utterances a user makes in a dialogue. This can be either typed input or the result of speech recognition, depending on what the medium of the dialogue is. The output of NLU is some representation of the meaning of the utterance that can then be used by the dialogue control module. The analysis should account for both the syntax and semantics of the utterance.

The syntactic analysis uses linguistic resources such as a lexicon and a grammar, both of which can be tailored for the domain of the dialogue. A typical representation of the syntactic structure of the utterance is a unification grammar, such as in Verbmobil [99]. A semantic representation can be computed using compositional semantics, in which the meaning of a sentence is constructed as a function of the meaning of its constituent parts. The meaning can then be encoded in a framework such as discourse representation theory, as we introduced in Section 2.2.1.

NLU in a dialogue context faces difficulties in addition to those in language processing in general. Dialogues often contain utterances that are not grammatically well-formed in terms of a normal sentence grammar. For instance, dialogue participants often use incomplete sentences or sentence fragments, or use self-repair when they speak. Speech recognition is not always reliable, and may not always output exactly the utterance that the user said. An NLU module must take this into account when analysing the utterance.

A solution is to make parsing more robust by restricting the vocabulary that the system understands. When the dialogue task is simple enough, keyword spotting is sufficient to extract the required information from the utterance. This is possible for instance in information seeking dialogues.

3.3.2 Domain Knowledge

A dialogue system will generally need to access some sort of external knowledge sources in order to complete its task. This could be for example a database for a timetabling application, or a link to a financial backend system for a banking application. A dialogue system can also access domain knowledge that aids its task, such as a knowledge base of task-related information. An example is the use of a planner to guide the task. BEETLE [101] uses a planner to reason about an electronics tutorial task, which in turn informs the dialogue control module. Similarly in TRAINS [31] a planner computes possible solutions for the collaborative task.

Enabling a dialogue system to communicate with outside knowledge sources allows it to abstract away from the task at hand and increases reusability and adaptability.

3.3.3 Natural Language Generation

The natural language generation (NLG) component of a dialogue system is responsible for generating the linguistic realisation of the system's dialogue move. In general, it must decide what content to express (although this may come from the dialogue control module), how to structure this information, and finally how it is realised, i.e. its surface form. The utterance string can then be passed on to a speech output module.

Possible solutions range from template based generation to complex language generation using linguistic resources. In the template based approach utterances are generated by mapping non-linguistic input directly to the linguistic surface structure [79]. This surface structure can contain gaps, and a well-formed utterance is generated when data has been inserted into each gap. This approach however limits the expressivity of the generation module, and can lead to unnatural sounding utterances when the same template is reused in a dialogue. Full natural language generation systems apply linguistic theory to the generation task. Instead of templates for linguistic realisation, syntactic and morphological knowledge is used to compute grammatically correct sentences. An example is a bidirectional grammar such as the Grammatical Framework [78], which treats generation as the inverse of parsing.

Language generation within a dialogue system must be able to relate utterances to the dialogue history, for example in order to generate correct and natural anaphors as a cohesive device. It also typically takes account of a user model, and must be equipped to generate not only full sentences, but also fragments, which often appear in dialogue.

3.3.4 Dialogue Strategy/Task Control

In order for all of these parts to function together properly, they need to be pulled together and organised by a central controlling module. The exact function varies depending on the needs of the particular system. At a minimum, it should fulfil the control role mentioned above.

The role of facilitating communication between modules is typically taken over by a dialogue manager. In addition, the dialogue manager can maintain a representation of the dialogue context in order to motivate further action. The control of the dialogue flow should be based on a theory of dialogue. This means that the dialogue manager can have a domain dependent plan which guides its action. It can also have knowledge of how to utilise modules in order to achieve its dialogue goal. This means interleaving the computations performed by the natural language understanding and generation with the access to other knowledge sources. The dialogue goal can be for instance a task, a tutoring goal, or information delivery.

3.4 Approaches to Dialogue Management

The job of the dialogue manager, as we saw in Section 3.3.4, is to control the flow of the interaction between the system and a dialogue participant based on some theory of dialogue. In this section we consider four types of design which support this role: finite state automata, the form-filling approach, agent-based systems, and the information state update approach.

Traum and Larsson [95] define dialogue management as consisting of the following functions:

- Maintenance of a dialogue context

- Providing context-dependent expectations for interpretation of observed signals as communicative behaviour

- Interfacing with task or domain processing, such as a database backend

- Deciding what content to express next

The approaches outlined here differ in their degree of support for each of these tasks. For instance, finite state systems have an implicit representation of dialogue context, whereas dialogue context is the central concept of the information state update approach. We now consider each approach in turn.

3.4.1 Finite State Automata

Systems which use the finite state automaton approach to dialogue management are characterised by a finite state machine which statically encodes all possible dialogues. A node represents a dialogue state or dialogue context, and an edge represents an action such as a dialogue move. An FSA-based dialogue manager can retain very tight control over the dialogue process. This type of dialogue manager also has a simple design and can be made deterministic. However, such systems are inflexible with respect to dialogue flow, and are not suited to supporting user initiative.

FSA-based dialogue managers are suited to simple tasks such as information elicitation, where a certain set of data must be collected by an agent in order to carry out some action. Contexts where the number of possible dialogues is small can be represented. Examples of finite state systems are the CSLU toolkit [65, 92], the DiaMant tool [35], or the Nuance demo banking system [71], which uses recursive transition networks.

3.4.2 Form-filling Approach

The form-filling approach (also known as the frame- or template-based approach) is more adaptable than finite-state. The system tries to incrementally fill slots in a form by asking the user questions. When enough information is present in the form the system can perform its task, such as a database lookup. A form-filling system works like a production system, where actions are determined by the current state of affairs.

The gain in flexibility over FSA systems is that the order in which slots in the form are filled is not strict, which allows some variation in the order in which information is elicited from the user. Overanswering can be dealt with, since more than one slot may be filled by a single user utterance. In example (5) we see that the user gives more information than was requested.

(5)   System: *What is your destination?*
        User: *London on Friday around 10 in the morning.*

Although the system has only asked the user for his destination in this timetable application, the system can recognise that three pieces of information (namely destination, day and time) have been supplied, and can insert these in the suitable slots of the form. A form-filling system is better equipped to handle user-initiative, and this ability can be strengthened by modelling the task as sets of forms, or contexts [3].

Form-filling is suited to situations in which the information flow is mainly in the direction of the system, for instance in timetable information systems, such as the Philips automatic train timetable information system [8]. VoiceXML [98] is an extension of the form-filling approach using an XML-based specification language to model dialogues. It supports both information-seeking and menu type dialogues.

3.4.3 Agent-based Approach

In dialogue systems using the agent-based design communication is viewed as an interaction of agents. Dialogue participants are represented by agents which can reason about actions and beliefs. Agent-based systems are suitable for mixed initiative dialogue because, for instance, the user can introduce new topics of conversation. Such systems can also use expectations to aid error correction. Due to the unconstrained nature of the interaction that agent-based systems support, there is a need for sophisticated natural language abilities. This contrasts with both finite state and form-filling systems, which restrict the language in which the interaction can take place. The Circuit Fix-it Shop [87] is an example of an

agent-based system which uses both a planner and a user model to facilitate collaborative problem solving dialogue.

3.4.4 Information State Update Approach

More recently the information state update (ISU) approach [95] has been developed within the TRINDI project [77]. It is motivated by the need to be able to formalise different theories of dialogue management to allow evaluation and comparison. From an engineering perspective, the ISU approach is motivated by the fact there are no hard and fast rules governing the design of dialogue systems, leading to bad support for reusability. The ISU approach proposes a unifying view of dialogue management based around the information state, in which domain independent theories can be implemented in a reusable foundation.

The information state of a dialogue is "the information necessary to distinguish it from other dialogues"[1]. It represents the cumulative effect of previous actions in the dialogue, and provides a context for future actions. It is similar to the "conversational score", "discourse context" or "mental state".

The ISU approach provides a method for specifying a theory of dialogue, and this theory has the following components:

- An *information state*

- *Representations* for the information state

- A set of *dialogue moves*

- A set of *update rules*

- An *update strategy*

We will now examine each of these in turn.

The information state is the description of the state of the discourse and its participants which is maintained by the dialogue manager. It stores dialogue level knowledge, such as the common context, linguistic and intentional structure, or aspects of beliefs or obligations, depending on what theory of dialogue it formalises. This context then forms the basis for the choice of action of the dialogue manager.

For each aspect of dialogue context that should be modelled, a representation must be chosen. This can range from a simple data structure like a list or a string to a more complex representation such as an attribute-value matrix, a term in a lambda calculus, or a discourse representation structure.

Dialogue moves, as introduced in Section 2.3.2, provide an abstraction away from utterances and other dialogue actions to a description of their function. When a dialogue move is performed its content may result in a change being made to the state of the dialogue.

---

[1][95], pg 3.

Which dialogue moves a dialogue theory includes is influenced by the theory itself and the domain of the dialogue.

As a dialogue progresses, the information state which describes it must be updated to reflect the effect that actions of the dialogue participants have on the dialogue context. How these updates take place is governed by update rules. Update rules fire in reaction to observed dialogue moves, and are specified by applicability conditions and effects. If the conditions are satisfied by the information present in the information state, then the effects of the rule can be carried out. The effects are changes that will be made to the information state. Thus update rules can be seen as transitions between information states.

In order to control how updates are made to the information state, an update strategy must be declared. This is an algorithm which decides which update rules should be allowed fire. Options for this algorithm include allowing the first applicable rule to fire, allowing all applicable rules to fire, or choosing between rules based on probabilistic information.

Systems applying the ISU approach

In this section we present some of the dialogue systems which have been implemented using the information state update approach as a theoretical framework. The TRINDI project and its follow up project SIRIDUS [76] have developed TrindiKit [94], a dialogue move engine toolkit, in order to provide a platform on top of which information state update based dialogue systems can be built. Using TrindiKit relieves the dialogue system developer of many software engineering issues, since components such as the information state types, dialogue move types and control modules will be similar from application to application.

A dialogue application built using TrindiKit consists of three layers. The bottom layer is TrindiKit itself, which provides the basic types, flow of control, and the software engineering glue required to build a system. The middle layer is a domain independent dialogue move engine. This is the implementation of a theory of dialogue. It is the part of the dialogue system which governs updates to the information state based on observed dialogue moves (the update module), and which chooses appropriate system dialogue moves (the selection module). The final layer then makes the application domain specific by adding domain resources and linguistic knowledge.

We now consider some systems built in the TrindiKit framework.

GoDis

One of the systems implemented using TrindiKit is GoDis [55], which implements issue based dialogue management. It is able to handle action and information-seeking dialogues, grounding and question accommodation. Question accommodation is accounted for according to Ginzburg's questions under discussion (QUD), introduced in Section 2.3.5.

The information state in GoDis is split into a shared and a private part. The shared part contains the information which is shared, or commonly believed, by the dialogue participants, and which has been explicitly established in the conversation. It contains the common ground of the dialogue, modelled as a set of propositions, the previous and

latest dialogue moves, issues, and the questions under discussion. The issues are all of the questions which have been raised in the dialogue. The QUD is a stack of questions which are locally under discussion.

The private part of the information state contains the system's beliefs, an agenda of short term intentions, a plan representing longer term dialogue goals, and a temporary slot used for storing tentative information. GoDis also uses a taxonomy of dialogue moves which includes six task moves (such as *ask*, *answer*), and a set of grounding moves [93].

Updates rules are used to perform question accommodation in order to integrate the content of utterances which are not on the QUD, issues, or plan of the system. For instance, the update rule **accommodateQuestion($\mathbf{Q}, \mathbf{A}$)** shown in (6) adds a question to the QUD whose answer the user has just uttered.

$$
(6) \quad
\begin{array}{ll}
\text{U-RULE:} & \textbf{accommodateQuestion}(\mathbf{Q}, \mathbf{A}) \\
\text{PRE:} & \left\{ \begin{array}{l} \text{in(SHARED.LU, answer(usr, } A)), \\ \text{in(PRIVATE.PLAN, findout}(Q)) \\ \text{domain :: relevant}(A, Q) \end{array} \right. \\
\text{EFF:} & \left\{ \begin{array}{l} \text{del(PRIVATE.PLAN, findout}(Q)) \\ \text{push(SHARED.QUD}, Q) \end{array} \right.
\end{array}
$$

The preconditions state that the linguistic meaning of the last utterance (LU) must have included the answer A, that the system must have in its plan the goal of finding out the answer to the question Q, and that A must be relevant to answering Q. When this is the case, the effects are carried out: the goal of finding the answer to Q is removed from the plan, and Q is added to the QUD. With this rule the accommodation of Q is not finished — another update rule will pop Q off the QUD and add A to the common ground.

GoDis uses an update strategy which begins by attempting to incorporate dialogue moves into the information state. It then repeatedly tries to integrate their effects, for instance performing accommodation. The system then continues with its current plan before removing, if possible, questions from the QUD which are no longer available.

Other Systems based on TrindiKit

A number of other systems have be implemented using TrindiKit as a framework. The EDIS system [63] uses a notion of information state based on grounding and discourse obligations [75, 74]. The common part of the information state includes a history of dialogue acts and a representation of the obligations of the dialogue participants. The information state also has a semi-private part, which contains discourse units which have been grounded. The private information includes the intentions of the dialogue participant being modelled.

MIDAS [17] is a system built on TrindiKit which uses DRT as its representational theory. DRSs are used to represent events mentioned in the dialogue and to handle grounding. TrindiKit is also the basis of BEETLE [28], which uses a layered architecture for managing tutorial dialogue. It reuses EDIS, adding update rules to account for tutoring dialogue moves such as hinting. It also adds a planning layer using O-Plan [30], which produces dialogue plans to guide the dialogue manager.

Dipper

Another system supporting the ISU approach to dialogue management is Dipper [18], a collection of software agents for prototyping spoken dialogue systems. It includes agents for speech input and output, dialogue management, and other supporting agents such as planners and natural language understanding. Dipper uses the Open Agent Architecture (OAA) [62], a framework for integrating software agents.

Dipper provides a dialogue management agent based on the information state update approach. It can be seen as a stripped down version of the TrindiKit implementation, and consists of two parts, both of which are represented by OAA agents. A dialogue move engine (DME) is responsible for dealing with input from other agents, the maintenance of the information state, and calling other agents. A DME server mediates between the DME and the other agents. Dipper's dialogue manager differs from the TrindiKit notion of information state update in that it has no explicit control algorithm. Instead control is determined solely by the design of the update rules, and the choice of what rule to fire is made on a first-come first-served basis. Also, Dipper does not separate update and selection rules, and the rule language is programming-language independent.

Dipper is for example currently being used in the implementation of the dialogue manager for the TALK project [91] and in the Witas project [56].

3.5 Summary

In this chapter we introduced the fundamentals of dialogue systems. After classifying some types of dialogue systems, including tutorial dialogue systems, we described the general architecture of a dialogue system, which is composed of natural language understanding, natural language generation, access to external knowledge sources, as well as a controlling module. The control module is typically encapsulated in a dialogue manager.

We continued with a review of four different approaches to dialogue management, namely finite state automata, the form-filling approach, agent-based systems, and the information state update approach. We covered information state update in some detail, outlining some systems which use this approach, as well as general frameworks such as TrindiKit and Dipper.

We will build on these relations in Chapter 4, where we will present the DIALOG project, which investigates mathematical tutoring using natural language dialogue. The tutorial dialogue system which has been built within the project supports flexible natural language and interfaces with a proof assistant. We follow this in Chapter 5 with a detailed presentation of the dialogue manager for the DIALOG demonstrator, which is an information state update based dialogue manager.

4

The DIALOG Project

4.1 Introduction

The DIALOG project [13, 72, 73] is a cooperative project between the research groups of Prof. Dr. Jörg Siekmann at the Department of Computer Science, and Prof. Dr. Manfred Pinkal at the Department of Computational Linguistics, both at Saarland University. The goal of the project is to investigate flexible natural language dialogue in mathematics, with the final goal of natural tutorial dialogue between a student and a mathematical assistance system.

In this chapter we give a high-level description of the project. We begin with an outline of the overall scenario in which the DIALOG project fits; that of a mathematical e-learning system. We follow this with a description of the corpus which was collected by the project. We describe the Wizard-of-Oz experiment, the corpus annotation, and mention some motivating phenomena that were found in the corpus. Finally we detail what role a dialogue manager has to play in the DIALOG system.

4.2 Scenario

An e-learning system for teaching mathematical proofs, for instance Activemath [68], involves many subsystems which support the tutoring of mathematics. A user model represents the experience, expertise and learning goals of the student. There is a concept of courses or lessons in order to structure learning over many sessions. Support for checking solutions and presenting mathematical content is provided by a backend source of structured mathematical knowledge. Finally the interface uses text, graphics, pointing and clicking to provide a natural way for the student to interact with the tutorial system.

At the interface level, work has been done showing that flexible natural language dialogue supports active learning [70], and this is where the DIALOG project fits into the e-learning scenario. The project investigates what requirements the flexible dialogue paradigm puts on a tutorial system, and how the different components of such a system must interact to combine mathematical tutoring with natural language dialogue.

DIALOG embodies multi-disciplinary research in a number of areas which include natural language understanding, tutorial dialogue, mathematical domain reasoning, and dialogue modelling.

A natural language understanding component must be able to analyse mixed formal and informal utterances, that is, sentences which contain both natural language and mathematical content. Based on the results of this analysis, the system should be able to determine the dialogue move corresponding to the utterance. Since DIALOG focuses on tutorial dialogue, a tutorial strategy must determine how the student is to be helped through exercises. The tutorial component guides the task aspect of the dialogue.

Mathematical domain reasoning must be provided by a mathematical database in order to analyse the mathematical content of the student's utterances, and to maintain a representation of the task that the student is solving. This also requires the use of automated theorem proving, encapsulated in a system known as an automated theorem prover (ATP). Mathematical domain reasoning in turn supports both natural language understanding and tutorial reasoning. Finally the system must model the dialogue in such a way that the different functions of the three components mentioned here can be integrated. This includes controlling the dialogue flow, linking to other systems, and maintaining a dialogue context.

These are the main research areas of the project. In order to implement and evaluate research results, a demonstrator system was additionally developed. This will be presented in Chapter 5.

4.3 Data Collection

When the DIALOG project began, little was known about the use of natural language in tutorial dialogue in the domain of mathematics, and the area was largely unstructured due to the lack of empirical data. The first phase of the project therefore included an experiment to collect a corpus of data. The experiment had two main goals. The first was to collect a corpus of student/tutor dialogues which would inform research in the areas of natural language understanding, tutoring and mathematical domain reasoning. The second goal was to annotate the corpus on three levels in order to investigate on the one hand the correlation between domain-specific content and its linguistic realisation, and on the other hand the correlation between the use, distribution and linguistic realisation of dialogue moves. A secondary goal was to test a newly-developed algorithm for socratic tutoring [34].

4.3.1   The Experiment

24 subjects with little to fair mathematical training took part in the experiment, in which they were asked to evaluate a tutoring system with natural language capabilities. The domain of mathematics which was used was naive set theory. It was chosen because it is a simple sub-domain of mathematics and is formally well understood, but proofs in the set theory still offer enough complexity to be used in a tutorial system. Before the dialogue session the subjects were given preparatory material explaining the concepts addressed in the proof exercises, and were asked to do a proof on paper. The session with the system involved proving the following three theorems[1]:

(7) $K((A \cup B) \cap (C \cup D)) = (K(A) \cap K(B)) \cup (K(C) \cap K(D))$

(8) $A \cap B \in P((A \cup C) \cap (B \cup C))$

(9) $When \ A \subseteq K(B), then \ B \subseteq K(A)$

The language of the dialogue was German, and subjects were told to prove the theorem stepwise and think aloud, rather than simply entering an entire proof. The intention of this was to encourage the subjects to build proofs incrementally and to use proof steps. After the tutoring session the subjects were given a second theorem to prove on paper, and a questionnaire.

The experiment was carried out in the Wizard-of-Oz (WoZ) paradigm, in which a mathematician playing the role of the wizard formulated the system responses. The experiment system used DiaWoZ [33], a Wizard-of-Oz support tool. Subjects were split into three groups in order to compare the minimal feedback, didactic and socratic strategies. The socratic group received hints from the wizard according to a socratic tutoring algorithm.

The resulting corpus [100] consists of 66 dialogues which contain 393 student utterances. It also includes audio and video recordings of each subject. An example of such a dialogue is soc20p[2], and a translation into English is shown in Figure 4.1.

4.3.2   Corpus Annotation

In order for the corpus to be used as a basis for research in DIALOG, a detailed analysis was necessary. Analysis of the corpus was done on the basis of annotation of utterances at three different levels.

The first is the linguistic level. The linguistic meaning of a sentence is its dependency-based deep semantics. The linguistic meaning was in turn annotated in terms of semantic dependency relations. Sentences were assigned tectogrammatical relations, such as CAUSE or CONDITION, to express the relation of a sentence to its dependent sentences.

The second annotation level is the dialogue level. The utterances were annotated with dialogue moves from the DAMSL taxonomy, which was extended for the annotation by

---

[1]In this and in subsequent examples, K refers to the complement operation, and P to powerset.

[2]The key "soc20p" encodes that this dialogue was conducted in the socratic group with participant number 20, and the exercise was the powerset proof. The original German text of this dialogue is given with annotation in Appendix A.

T1:   Please show: $A \cap B \in P\ (\ (\ A \cup C\ ) \cap (\ B \cup C\ )\ )$!
S1:   It holds that: $(\ A \cup C\ ) \cap (\ B \cup C\ ) = C \cup (\ A \cap B\ )$
T2:   Correct.
S2:   From this we get: $P\ (\ (\ A \cup C\ ) \cap (\ B \cup C\ )\ ) = P\ (\ C \cup (\ A \cap B\ )\ )$
T3:   Yes, that follows.
S3:   And for the powerset it holds : $P\ (\ C \cup (\ A \cap B\ )\ ) = P\ (\ C\ ) \cup P\ (\ A \cap B\ )$
T4:   Do you really mean : $P\ (\ C \cup (\ A \cap B\ )\ ) = P\ (\ C\ ) \cup P\ (\ A \cap B\ )$?
S4:   No, i think : $P\ (\ A\ ) \cup P\ (\ B\ ) = P\ (\ A \cup B\ )$
T5:   That's not correct! Maybe you should read up in your study material.
S5:   Sorry, it holds of course : $P\ (\ C \cup (\ A \cap B\ )\ ) \subseteq P\ (\ C\ ) \cup P\ (\ A \cap B\ )$
T6:   Really?
S6:   Oh, no... the other way around
T7:   That's correct at last!
S7:   So, it holds: $P\ (\ C\ ) \cup P\ (\ A \cap B\ ) \subseteq P\ (\ C \cup (\ A \cap B\ )\ )$ and $A \cap B \in P\ (\ A \cap B\ )$ from which the proof follows
T8:   That's correct. I'll sum up again: We want to prove that $A \cap B \in P\ (\ (\ A \cup C\ ) \cap (\ B \cup C\ )\ )$ holds. We see that $P\ (\ (\ A \cup C\ ) \cap (\ B \cup C\ )\ ) = P\ (\ (\ A \cap B\ ) \cup C\ ) \supseteq P\ (\ C\ ) \cup P\ (\ A \cap B\ ) \supseteq P\ (\ A \cap B\ )$. But since $A \cap B \in P\ (\ A \cap B\ )$ holds, the assumption is proved.

Figure 4.1: Dialogue soc20p from the corpus.

adding a task dimension. The taxonomy is tailored to account for the types of moves found in tutorial dialogues, as well as the management of tutorial dialogue in general. Dialogue moves consist of 6 dimensions:

Forward-looking This characterises the effect an utterance has on the subsequent dialogue.

Backward-looking This dimension captures how the current utterance relates to the previous discourse.

Task In contrast to the DAMSL design, here the task content of an utterance constitutes a separate dimension. It captures functions that are specific to the task at hand and its manipulation. This dimension is particularly important for the genre of tutorial dialogues, and has itself an inner structure.

Communication management This concerns utterances that manage the structure of the dialogue, for instance to begin or end a subdialogue.

Task management This dimension captures the functions of utterances that address the management of the task at hand, for instance beginning a case distinction or declaring a proof complete. It has two sub-dimensions, namely proof task and tutoring task, which are used to annotated student and tutor utterances respectively.

Communicative status This dimension concerns utterances which have unusual features, such as non-interpreted utterances.

The full taxonomy of dialogue moves is presented in [96].

The third annotation level is the tutoring level. Using a taxonomy of hints, the annotation categorises what type of hint the tutor has given, for instance whether the hint was active or passive, or what domain concepts the hint addressed. Proof steps in the students' utterances were annotated to reflect the category of the student answer. The annotations describe for instance the accuracy, completeness and relevance of the proof step.

4.3.3 Phenomena in the Corpus

The result of the annotation of the corpus was a list of key phenomena which are found in the area of tutorial dialogue on mathematical proofs. We can divide these into three distinct levels.

The first is linguistic, and is detailed in [14]. Here we found evidence of varying degrees of formal content. Concepts were referred to sometimes with a linguistic expression, e.g. "element of", and sometimes with a mathematical symbol, e.g. "$\in$". We also found a tight interleaving of natural language with formulae, for instance "B contains no $x \in A$". There was much ambiguity in reference to concepts, such as in "$(A \cup B)$ must be in $P((A \cup C) \cap (B \cup C))$", where the word "in" could mean the element or the subset relation. There were also informal references to mathematical knowledge.

At the second level, the tutorial level, we found mixed effectiveness of didactic and socratic tutoring methods. This was measured based on the proofs done on paper that the subjects were asked to do before and after the tutorial session. Subjects in the didactic group were found to have learned the most, whereas the socratic group learned less than expected.

The third level is domain reasoning. Here we found evidence of much underspecification of mathematical concepts and statements. This means that references to axioms or inference rules which were part of the proof step were simply omitted. The intended proof step direction was also often not explicitly stated. There was a very mixed granularity of proof steps, with some students making many low-level steps and others larger higher-level proof steps. Subjects also mainly stated conclusions of rule applications rather than giving the applications themselves. One phenomena related to ambiguity on the linguistic level is non-coreference of mathematical symbols, as illustrated in (10).

(10) DeMorgan rule 2 says: $K(A \cap B) = K(A) \cup K(B)$. In this case e.g. $K(A) =$ the term $K(A \cup B)$, $K(B) =$ the term $K(C \cup D)$.

Here the two occurrences of A in the utterance "$K(A) =$ the term $K(A \cup B)$" are clearly not intended to corefer. We also found that an alternative view of the notion of proof is needed, namely a human-oriented, incrementally built proof in which assertion level reasoning [50] plays an essential role.
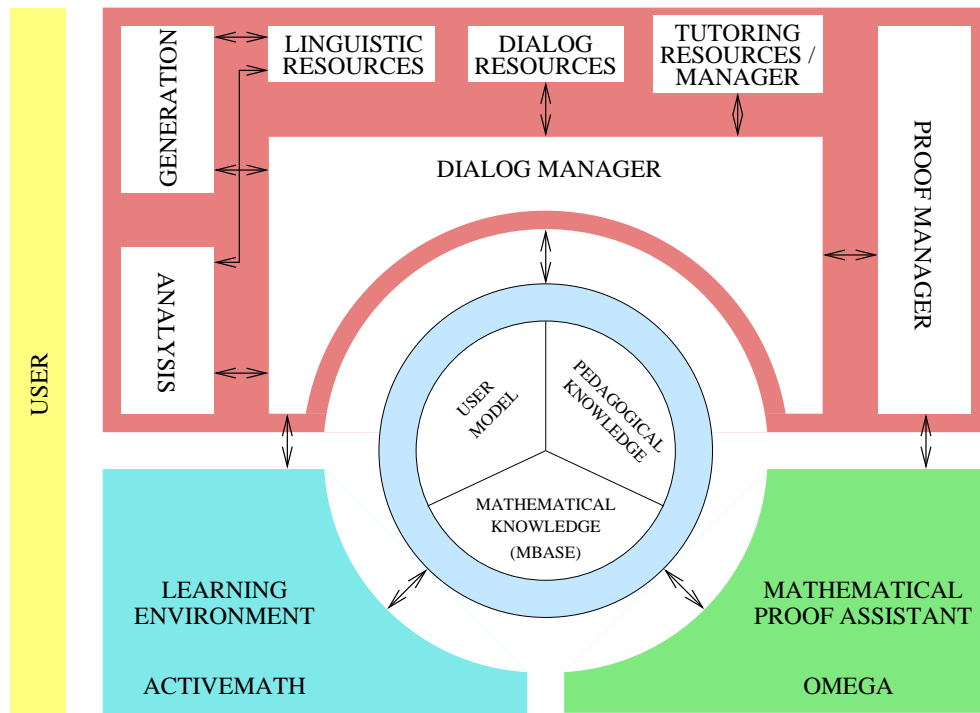
Figure 4.2: The architecture of the DIALOG system.

All in all, these phenomena indicate that a close interplay of natural language under-standing, mathematical reasoning, tutorial reasoning, and dialogue modelling is required in order to accurately model mathematical tutorial dialogues.

4.4 The Role of the Dialogue Manager

In this section we motivate and describe the role that the dialogue manager has to play in the DIALOG system. As motivation, we will briefly consider the architecture of the system. The two functions that the dialogue manager must fulfil are the provision of inter-module communication and the maintenance of the dialogue context. To achieve this functionality, the dialogue manager will use the information-state update approach to dialogue management.

The high-level architecture of the DIALOG system is shown in Figure 4.2, with the core modules shown in the upper half of the diagram. The natural language understanding aspect of the project is implemented in the analysis module, which draws on linguistic resources such as a domain oriented lexicon and a grammar. Linguistic resources are also employed by the generation module, which verbalises the dialogue move to be performed by the system. Tutorial issues are handled by the tutoring manager, which comprises for instance the socratic tutoring algorithm. The proof manager is responsible for domain

reasoning, including the evaluation of the student's proof step with respect to the partial proof that the student has built so far. It mediates between the Dialog system and the mathematical assistance system Ωmega. The function of each module is presented in more detail in Section 5.3.

4.4.1   Module Communication

We have seen in Section 4.3.3 that the interleaving of many system modules, such as natural language analysis and domain reasoning, is necessary to account for many phenomena found in tutorial dialogue on mathematics. The dialogue manager is the part of the Dialog system that allows this interplay to take place. As we see from the architecture of the system, the dialogue manager has a central role to play in facilitating communication, or message passing, between the many system modules. System modules are not able to communicate directly with each other, rather their communication takes place in a star-like design with the dialogue manager at the hub. The motivation behind this design is that the dialogue manager can act as the mediator of all system communication, and thus is able to control module execution.

4.4.2   Maintenance of the Dialogue Context

We have seen that one of the functions of the dialogue manager is to maintain the dialogue context. Such a representation of the current state of the dialogue is necessary in order to motivate system action. For instance, the dialogue context forms the basis for computing the system's dialogue move because it contains, among other things, the user's dialogue move, and the linguistic meaning of the utterance.

In the Dialog project scenario, the system modules shown in Figure 4.2 must be able to access information that describes the current state of the dialogue. By storing this information centrally, the dialogue manager can make it available to modules on demand. Central storage also supports information exchange. This is useful for the natural language analysis module and the domain reasoning module, since they can ideally use each other's results to support their own analysis.

The information stored in the dialogue context depends on the application. In the case of Dialog, we want to include at least an utterance history, and representations of both the user's and the system's dialogue move, as well as the linguistic meaning of the user's utterance. In addition, the dialogue context will contain information that is shared between system modules, such as the current tutorial mode or the evaluation of the proof content of the user's last utterance.

4.4.3   Design

Now that we have outlined what role the dialogue manager should play in the Dialog system, we can consider which of the approaches to dialogue management presented in Section 3.4 is most suitable. Both the finite state approach and the form-filling approach

do not offer the required flexibility for the DIALOG project. Finite state methods are not suited to flexible dialogue flow, and forms, although the dialogue flow is more adaptable, are not sophisticated enough to handle two-way the information exchange required by the task of tutoring of mathematical proofs. Both of these features are critical in DIALOG.

We have decided to use the information state update approach to dialogue management. In the information state update approach the notion of information state and its representation can be adapted to the dialogue task. This allows us to have a sophisticated representation of both the dialogue context and domain related information. The use of a central information state also supports the sharing of dialogue-level information between system components, which is essential to interleave their computation.

From the annotation of the corpus, we have a detailed notion of dialogue move, and the information state approach will allow us to directly integrate these. Dialogue moves motivate the design of the update rules in the system and are part of the dialogue resources that the dialogue manager uses.

4.5 Summary

In this chapter we have presented the DIALOG project and the corpus which was collected to inform the research. We then considered what function the dialogue manager in DIALOG has, motivated by the findings from the analysis of the corpus and the general project goals.

In the next chapter we will show how the results of the research done in each of the areas we have introduced here have been implemented in a demonstrator. There we will see in detail how the dialogue manager interacts with and facilitates the system modules.

5

The DIALOG Demonstrator

5.1 Introduction

In this chapter we present the dialogue manager built for the DIALOG demonstrator. The dialogue manager is the first contribution of this thesis, and has been documented in [20, 21]. We begin with an overview of the demonstrator, including a description of its architecture and the function that the dialogue manager has. We then detail each of the system modules in turn[1], giving examples of the input to and output from each for a typical turn in the dialogue which was presented by the demonstrator.

In section 5.4 we introduce the Rubin tool, a platform for building dialogue management applications based on the information state update approach. We then present the dialogue model, which is the specification of the behaviour of the dialogue manager. The dialogue model is written by a developer in order to create a dialogue manager using the Rubin platform. Finally we discuss some issues raised in the development of the demonstrator which will form part of our motivation later in the thesis.

The DIALOG demonstrator was developed to illustrate the functionality of the DIALOG system at hand of a few dialogues from the project's Wizard-of-Oz corpus. We concentrated on dialogue did16k, which is given with full annotation in appendix B. The task that the student is asked to prove is the theorem in (11), where K stands for the complement operation.

(11) $K((A \cup B) \cap (C \cup D)) = (K(A \cup B) \cup K(C \cup D))$

This dialogue exhibits the phenomena outlined in Section 4.3.3 that were found in the DIALOG corpus. This means it demands that all modules collaborate to complete the dia-

---

[1]The individual modules of the demonstrator system were developed by other researchers in the DIALOG project, but are documented here to motivate the design of the dialogue manager.
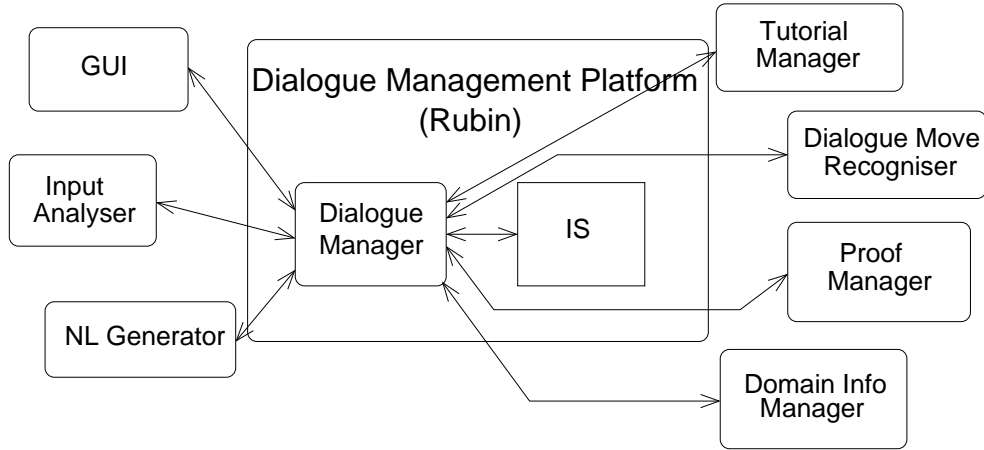
Figure 5.1: Architecture of the Dialog Demonstrator.

logue, involving natural language understanding, proof management, tutorial management and natural language generation. The examples in this chapter of information exchange between modules are all taken from this sample dialogue.

5.2   Overview of the Demonstrator

5.2.1   System Architecture

Figure 5.1 shows the architecture of the demonstrator. We see that this differs from the architecture shown in Figure 4.2, since some modules have been added and some, such as the learning environment, were not included in the demonstration. The new modules are the domain information manager, which provides mathematical knowledge for the tutorial component, and the dialogue move recogniser, which computes the dialogue move of an utterance based on its linguistic meaning. Each of the modules interface directly with the dialogue manager, which in turn can access the information state. The dialogue manager and the information state are implemented on top of the Rubin dialogue management platform.

Here we would like to stress that there are strictly speaking two different notions of a dialogue manager, depending on what is seen to be its main task. One is that a dialogue manager has the function of computing a dialogue move based on the partial dialogue leading up to the current move, and the contents of the information state. The other notion of a dialogue manager is a platform which supports the development of a dialogue-based application. In this sense the dialogue manager provides features such as communication between modules, an information state, and a language to define update rules, etc. This is the approach described in this section. The Dialog demonstrator contains subsystems which fulfil both of these tasks, and in this thesis we are dealing with the second notion of

dialogue management — the development platform for dialogue applications.

5.2.2   The Function of the Dialogue Manager

We have seen that one of the functions of the dialogue manager is to act as the communication link between modules. Modules are not able to pass messages directly to each other, for design as well as technical reasons. The design of the system is such that the dialogue manager is the mediator of all communication between system modules, and in this way is able to control all message passing and thus the order of module execution. Each result computed by a module needs to be stored in the information state. Since the dialogue manager receives the results of each module's computation, it has the opportunity to immediately make the corresponding information state update, and has full control of top-level system execution.

On the technical side, the design of the system in Figure 5.1 shows that it is a star type architecture. Each module is connected only to the central server (the dialogue manager) and there is no link between modules themselves. The result of this is that all information must first be sent to the dialogue manager, where it can be stored in the IS, and is then passed on to the modules that require it.

5.2.3   Dialogue Move Selection

What dialogue move the system produces is determined based on information supplied by each of the modules. We now outline how the system completes a single turn, that is, how each module plays its role in computing the system's utterance.

The first source of information is the content of the user's utterance. This comes from the input analyser in the form of linguistic meaning of the utterance and the proof step it contains, and from the dialogue move recogniser, which determines the dialogue move representing the utterance. The linguistic meaning can impose obligations on the system; for instance if the user poses a question, the system should create a dialogue move which answers the question, thereby discharging the obligation.

In order to decide on the mathematical content of its reply, the system combines information from the proof manager, the tutorial manager and the domain information manager. Given the proof step that the user's utterance contained, the proof manager determines whether in the context of the proof that the user is constructing the proof step is correct, if it has the appropriate level of detail, and if it is relevant. With this information the dialogue manager can decide for example to confirm a correct step, signal incorrectness, or ask the tutorial manager to add a hinting aspect to the response dialogue move. The tutorial manager contributes the whole task dimension of the system's dialogue move. This may include a hint, which is typically to supply to the user or elicit from the user a mathematical concept (given by the domain information manager) that should help the user progress in the current proof state.

The final step is to pass the now complete system dialogue move, along with any extra specifications required, to the generation module to be verbalised. The resulting utterance

Figure 5.2: The DiaWoz tool, showing the first five moves of the dialogue.

is output to the GUI, and the turn passes to the user. At this point the system waits for the next user utterance to be received. The result is a sequence of dialogue moves according to the model of the dialogue.

5.3 System Modules

In this section we detail the functions of each of the seven modules which are connected to the dialogue manager. Information enclosed in chain brackets represents a structure, information in round brackets is a list. Each of the examples of input and output to or from a module relates to the computation involved in responding to the student utterance "Nach deMorgan-Regel-2 ist $K((A \cup B) \cap (C \cup D)) = (K(A \cup B) \cup K(C \cup D))$". The notions of input and output in this section depend on point of view: the results that a module computes are its output, which then become the input to the dialogue manager. In this section we take the point of view of the respective module. Input is the data which it receives from the dialogue manager, and output is the result of its computation which is then sent back to the dialogue manager.

5.3.1 Graphical User Interface

The GUI of the demonstrator program is an extension of the DiaWoZ tool [33], which has been developed at the very beginning of the DIALOG project to support the Wizard-of-Oz experiments in which we collected our corpus. The GUI is illustrated in Figure 5.2. In the lower text field the user types his input, which when submitted, appears in the upper

text field, or conversation field. System utterances also appear in this field. At the top of the GUI is a row of buttons for mathematical symbols which do not typically appear on a keyboard. On the right there are two extra buttons and an input field. These are used to set the tutorial mode, i.e. minimal feedback, didactic or socratic, and to delete the last turn. They were added to allow the demonstrator to show the full functionality of the system within a single sample dialogue. The GUI is implemented in Java.

Input A string (the system utterance), which is then displayed in the conversation field, e.g.: "Das ist richtig!"

Output The user utterance (st_input), the tutorial mode if it was set since the last user utterance (mode), and a Boolean flag (delete) indicating whether a deletion of the last turn is to be carried out:

$$\{ \quad \text{st\_input} \quad = \quad \text{"Nach deMorgan-Regel-2 ist } K((A \cup B) \cap (C \cup D)) = (K(A \cup B) \cup$$
$$K(C \cup D))\text{"}$$
$$\text{mode} \quad = \quad \text{"min"},$$
$$\text{delete} \quad = \quad \text{false} \}$$

5.3.2 Input Analyser

The input analyser receives the user's utterance and determines its linguistic meaning and proof content. Input is syntactically parsed using the OpenCCG parser [12], and its linguistic meaning is represented using *Hybrid Logic Dependence Semantics* (HLDS) [11].

Input The user's utterance as a string (see st_input in the output of the GUI above).

Output A structure containing the linguistic meaning (LM) represented in HLDS and the underspecified proof step contained in the utterance, in an ad-hoc LISP-like representation. The language is abbreviated with LU, which stands for proof language with underspecification. This is a language in the spirit of the proof representation language described in [10], but designed for the inter-module communication requirements of the DIALOG project:

```
{   LM   =   @h1(holds ∧ <CRITERION>(d1 ∧ deMorgan-Regel-2) ∧ <PATIENT>(f1 ∧ FOR-
                MULA))
    LU   =   (input (label 1_1)
                    (formula (= (complement (intersection (union a b)
                                                          (union c d)))
                             (union (complement (union a b))
                                    (complement (union c d)))))
                    (type ?)
                    (direction ?)
                    (justifications (just
                      (reference demorgan-2)
                      (formula ?)
                      (substitution ?)
                      (role:from))))
             }
```

5.3.3   Dialogue Move Recogniser

The dialogue move recogniser determines the values of the six dimensions of the dialogue
move associated with the user's utterance. It does this based on the linguistic meaning
output by the input analyser.

Input   The linguistic meaning of the user's utterance, which is the LM element in the
output of the input analyser.

Output   A dialogue move or set of dialogue moves corresponding to the student's utter-
ance:
    {   fwd = "Assert",
        bwd = "Address_statement",
        commm = "",
        taskm = "",
        comms = "",
        task = "Domain_contribution" }

   This dialogue move encodes the student's utterance in the forward-looking (fwd),
backward-looking (bwd), and task (task) dimensions. "Assert" in the forward dimension
means that the speaker has made a claim about the world, and introduced an obligation
on the hearer to respond to the claim. In the backward dimension, "Address_statement"
means simply that the utterance addresses a preceding statement, here the statement which
posed the problem at hand to the student. The task dimension "Domain_contribution"
describes a dialogue move which is concerned with resolving the domain task for the ses-
sion. In this case, the utterance is a domain contribution because the student proposes to
apply the de-Morgan rule, and in doing so contributes to the task of building a proof.

5.3.4   Proof Manager

The proof manager is the mediator between the dialogue manager and the mathematical proof assistant Ωmega–Core [85, 86]. The proof manager replays and stores the status of the partial proof which has been built by the student so far, and based on this partial proof, it analyses the soundness and relevance of a next proof step. It also investigates, based on a user model, whether the proof step has the appropriate granularity, i.e., if the step is too detailed or too abstract, and whether it is relevant. The proof manager also tries to resolve ambiguity and underspecification in the representation of the proof step uttered by the student. In doing this the proof manager ideally accesses mathematical knowledge stored in MBase [54] and the user model in ActiveMath [68], and also deploys a domain reasoner, usually a theorem prover. These tasks for the proof manager are very ambitious; some first solutions are presented in [10, 51].

The proof manager receives the underspecified proof step which was extracted from the user's utterance by the input analyser. This is encoded in the proof representation language LU [10] (LU in the output of the input analyser). The proof manager is able to reconstruct the proof step that the student has made using mathematical knowledge, its own representation of the partially constructed proof so far and the potentially underspecified representation of the user proof step. It then outputs the fully specified representation of the user proof step, along with the step category, (e.g. correct, incorrect, irrelevant, etc) and whether the proof was completed by the step. It also includes a number of possible completions for the proof that the student is building (stored in `completeProofs`). This is used by the domain information manager and the tutorial manager to determine what mathematical concept to either give away to or elicit from the student.

Input   The underspecified proof step output by the input analyser (LU in Section 5.3.2).

Output   An evaluation of the proof step.

```
((KEY 1_1) -->
   ((Evaluation
     (expStepRepr
      (label 1_1)
      (formula (=(complement(intersection(union(A B) union(C D)))
                union(complement(union(A B)) complement(union(C D))))))
      (type inference)
      (direction forward)
      (justification (
       (reference demorgan-2)
       (formula nil)
       (substitution ((X union(A B) Y union(C D))))
       (role nil))))
    (StepCat correct)))
```

```
(ProofCompleted false)
(completeProofs ....))
```

This example shows the similarity of the proof manager's output to the underspecified proof step that it receives from the input analyser. In this case, the proof manager was able to resolve a number of underspecified elements of the proof step, namely type, direction and substitution. It was also able to determine that the proof step was correct (the StepCat item), and added "ProofStepCompleted false", meaning that after this proof step has been integrated into the student's partial proof plan, the proof is still not complete.

5.3.5   Domain Information Manager

The domain information manager determines which domain information is essentially addressed in the attempted proof step and assigns the value of the domain information to the expected proof step specified by the proof manager. It receives both the underspecified and evaluated proof step in order to categorise the user input in more detail.

Input   The proof step from the input analyser and its evaluation from the proof manager.

Output   Proof step information:

{	domConCat:	"correct",
	proofCompleted:	false,
	proofstepCompleted:	true,
	proofStep:	"",
	relConU:	true,
	hypConU:	true,
	domRelU:	false,
	iRU:	true,
	relCon:	"∩",
	hypCon:	"∪",
	domRel:	"",
	iR:	"deMorgan-Regel-2"}

5.3.6   Tutorial Manager

The job of the tutorial manager is to use pedagogical knowledge to decide on how to give hints to the user [34], and this decision is based on the proof step category (correct, irrelevant, etc), the expected step, a naive student model and the domain information used or required. The tutorial manager can decide for instance to elicit or give away the right level of information, e.g., a mathematical concept, or to simply accept or reject the proof step in the case that it is correct or incorrect, respectively. This decision is influenced by the tutorial mode, which can be "min", for minimal feedback, "did", for a didactic tutorial

strategy, in which answers and explanations are constantly provided by the tutor, or "soc" for socratic, where hints are used to achieve self-explanation.

Input The tutorial mode, the task dimension of the user's dialogue move, which is determined by the dialogue move recogniser, and the proof step information, which is the whole output from the domain information manager. This includes the evaluation of the user's proof step, and the possibilities for the next proof step, according to the proof manager.

Output A tutorial move specification, that is, the tutorial mode and the task content of the system dialogue move.

```
{   mode   =   "min";
    task   =   (signalAccept;
                {proofStep= ""; relCon= ""; hypCon= ""; domRel= ""; iR= "";
                taskSet= ""; completeProof= ""})
                }
```

The task dimension captures functions that are particular to the task at hand and its manipulation. That is, it encodes aspects of a dialogue move that talk "about" the theorem proving process, since this is the task in a mathematical tutorial dialogue. Here the task dimension value is "signalAccept", which confirms the correctness of a domain contribution, and which ultimately leads to the system utterance "Das ist richtig!" ("That's right!").

The remaining values in the task dimension are parameters for different hint categories, a subset of which was used for the demonstrator. For each of the hint categories (which are defined in a domain ontology [96]) certain parameters are passed to the generation module. When a proof step is to be given away, the value of the parameter proofStep is the formal proof step. Similarly for a relevant concept (relCon) or a hypotactical concept (hypCon). domRel refers to a domain relation which is to be mentioned in a hint, and iR is an inference rule (such as a DeMorgan Law). The task which was set for the tutorial session is stored in taskSet, and completeProof contains a representation of the complete proof that the user has built. This is used for example when a recapitulation is given at the end of a tutorial dialogue. In this example each parameter has an empty string as its value because the task dimension move "signalAccept" does not need any parameters. It simply expresses confirmation that the last user proof step was correct.

5.3.7   NL Generator

The natural language generation system used in DIALOG is *P.rex* [32]. *P.rex* is designed to present complete proofs in natural language, and thus a number of aspects had to be adapted for the DIALOG project. In a dialogue setting utterances are produced separately and sequentially, not as a complete coherent text. Also, referents of anaphors are constantly changing as the dialogue model develops. As well as this, *P.rex* was designed for English language generation, and the DIALOG system conducts dialogues in German.

The NL Generator receives a dialogue move and returns an utterance whose function captures each dimension of the move.

Input A system dialogue move specification, that is, a six-dimensional dialogue move along with the current tutorial mode, e.g.:

```
{  mode      =   "min";
   fwd       =   "Assert";
   bwd       =   "Address_statement";
   task      =   ( "signalCorrect", {proofStep= "", relCon= "", hypCon= "",
                   domRel="", iR= "", taskSet= "", completeProof= ""});
   comms     =   "";
   commm     =   "";
   taskm     =   ""}
```

The value of the task dimension of the dialogue move and the tutorial mode is taken from the output of the tutorial manager. The other 5 dimensions are computed by the dialogue manager itself, based on the dialogue move of the student's utterance. For instance, the "Address_statement" in the backward-looking dimension is in response to the "Assert" in the forward-looking dimension of the student's dialogue move.

Output The natural language utterances that correspond to the system dialogue moves. These then become the input to the GUI, e.g. "Das ist richtig!".

5.4   Rubin

To develop the demonstrator we used Rubin [36], a platform for building dialogue management applications developed by CLT [89]. It uses an information state based approach to dialogue management, and allows quick prototyping and integration of external modules (called "devices"). The developer of a dialogue application writes a dialogue model describing the dialogue manager, which is then able to handle device communication, parse and interpret input, fire input rules based on messages received from clients, and execute dialogue plans. In this section we give a formal view of Rubin's dialogue model.

5.4.1   The Rubin Dialogue Model

The Rubin term "dialogue model" refers to a user–defined specification of system behaviour. It should be noted that this does not refer to the model of domain objects, salience, and discourse segments, etc, as in other theories of discourse. It is defined according to the following grammar:

$$
\begin{aligned}
dialogue\_model \quad &:= \quad IS \\
&\qquad device\_declaration* \\
&\qquad [grammar] \\
&\qquad support\_function* \\
&\qquad plan* \\
&\qquad input\_rule*
\end{aligned}
$$

In the following sections we detail each part of the dialogue model grammar in turn.

Information State

The information state in Rubin is implemented as a set of freely–defined typed global variables (called slots) which are internally visible in the dialogue manager. Slots can have any of Rubin's internal data types: `bool`, `int`, `real`, `string`, `list` or `struct`. The IS is specified by the following syntax:

$$
\begin{aligned}
IS &:= slot^* \\
slot &:= label\ [\ :\ type\ ][\ =\ value] \\
type &\in \{\texttt{bool},\ \texttt{int},\ \texttt{real},\ \texttt{string},\ \texttt{list},\ \texttt{struct}\}
\end{aligned}
$$

where *label* is any variable name, and *value* is an object which has the correct type in its context, e.g. a quoted string for a variable of type `string`. `list` and `struct` objects are specified as follows:

$$
\begin{aligned}
list &:= [\ ]\ |\ [\ value\ \{,\ value\}^*\ ] \\
struct &:= \{slot^*\}
\end{aligned}
$$

For a slot of type `struct`, it is possible to either directly specify the slot as having the type `struct`, or to specify the exact structure of slots within the struct, for example:

$$
location\ :\ \{\ city : string \\
airport : string\ \}
$$

External Devices

Arbitrary modules that send and receive data can be connected to the Rubin server, for example a speech recogniser or a graphical user interface. A connection is specified by a unique device name and a port number over which communication takes place:

$$
device\_declaration\ :=\ device\_name : port\_number\ ;
$$

Connecting a module as a device is described in Section 5.4.3.

Grammar

Using a grammar written in the Speech Recognition Grammar Format (SRGF), it is possible to preprocess (i.e. parse and interpret) natural language input from a device before performing further computations within the dialogue manager, or sending the input to another module. The grammar is context-free with semantic tags. It takes a string as input and returns either the corresponding semantic tagging, or the string which was recognised, if no semantic tags are given.

For instance, a grammar could be used to parse a natural language utterance containing the time of day before sending the utterance to a sentence analysis module for further processing. In this case a grammar would parse strings like "four fifteen p.m." or "a quarter past four" and determine a semantic representation such as:

$$
\{\ h = 16\ ,\ m = 15\ \}
$$

Support Functions

Auxiliary functions can be defined in Rubin for use within the dialogue manager, and these are globally visible. These can perform operations on the internal data types used in the dialogue model, and the syntax is nearly identical to ANSI C:

$$support\_function \quad := \quad \{type \mid void\}\ name(\{type\ label\}^*)\ \{statement^*\ \}$$

where the first occurrence of *type* is the return type of the function, *name* is a label which begins with a small letter, the *label*s are the arguments of the function, and a *statement* is a C-style statement, including assignment, variable declaration, and constructs such as if, while, etc. Statements can also set the value of slots in the information state, and make calls to devices.

Plans

These are special functions with return type boolean. A plan has positive and negative preconditions which are tested for the duration of its execution. If at any point a positive precondition is fulfilled, execution is interrupted and the plan returns true. This is used when the goal of a plan is to elicit some piece of information; when that piece of information is found, the plan exits successfully. If a negative precondition evaluates to true, execution is interrupted and the plan returns false. Plans are defined according to the following syntax:

$$
\begin{aligned}
plan \quad &:= \quad name(\{type\ label\}^*) \\
&\qquad preconditions \\
&\qquad \{statement^*\ \} \\
preconditions \quad &:= \quad [\ ] \mid [\ precondition\ \{,\ precondition\}^*\ ] \\
precondition \quad &:= \quad pos\_precon \mid neg\_precon \\
pos\_precon \quad &:= \quad :\ condition \\
neg\_precon \quad &:= \quad !:\ condition \\
condition \quad &:= \quad slot\_name\ \{\ == \mid != \ \}\ value
\end{aligned}
$$

Here a *statement* is similar to a statement in a support function. It can make changes to the IS and call other devices.

Input Rules

These are rules which carry out arbitrary actions based on input from devices connected to the dialogue manager, and are specified with the following syntax:

input_rule	:=	*IS_constraints* {__|*device_name*} *input_pattern* : {*statement**}
IS_constraints	:=	__ | {*matching**}
input_pattern	:=	__ | *label* | *listpattern* | *structpattern*
listpattern	:=	[ ] | [*pattern* {, *pattern*}*]
structpattern	:=	{*pattern* {, *pattern*}*}
pattern	:=	*matching* | *label*
matching	:=	*slot* = *value*

When input is received from some device a rule can be fired based on the content of the fields in its header. *IS_constraints* is a set of constraints (which may be empty) on values in the IS which must hold for the rule to fire. That is, for the constraint

```
{ x = 3 }
```

the value of the slot x in the information state must be 3 for the rule to fire. *device_name* must be the same as the unique name of the device from which the input came. If "__" is given as the device name, the rule can match input from any device. The *input_pattern* must match[2] with the input from the device. A side effect of this matching is that the input becomes bound to the variables which are implicitly declared in the input pattern. For example, the rule

```
_, "SA", { LM = typeof_lm, LU = input} : {...}
```

will only match on input from the device called "SA" with input of type struct, where the structure contains 2 slots, LM and LU. This rule puts no constraints on the type of the values in these two slots. When the rule fires, the values in the slots are bound to the labels typeof_lm and input respectively, and these labels are visible in the body of the rule. The first rule in the dialogue model whose IS constraints, device name and input patterns match is executed.

Rule bodies are just inlined plans that can update the IS, push other plans, etc. Thus given a data object as input, a rule can make changes to the IS, to the plan stack, or to both.

In general an input rule denotes a function f:

$$f \in AS \times PS \times Inputs \to AS \times PS$$

where

AS	=	set of all assignments of IS slots
PS	=	set of all possible states of the plan stack
$Inputs$	=	the set of Rubin data objects

An input rule $f(as, ps, input)$ may fire when as is an assignment of information state slots which satisfies the IS constraints of the rule and *input* matches with the input pattern of the rule.

---

[2]Here we speak of matching as opposed to unification. It is not possible to have variables as the values of slots in the information state, so matching is sufficient to decide on the applicability of rules and to bind input to local variable names.
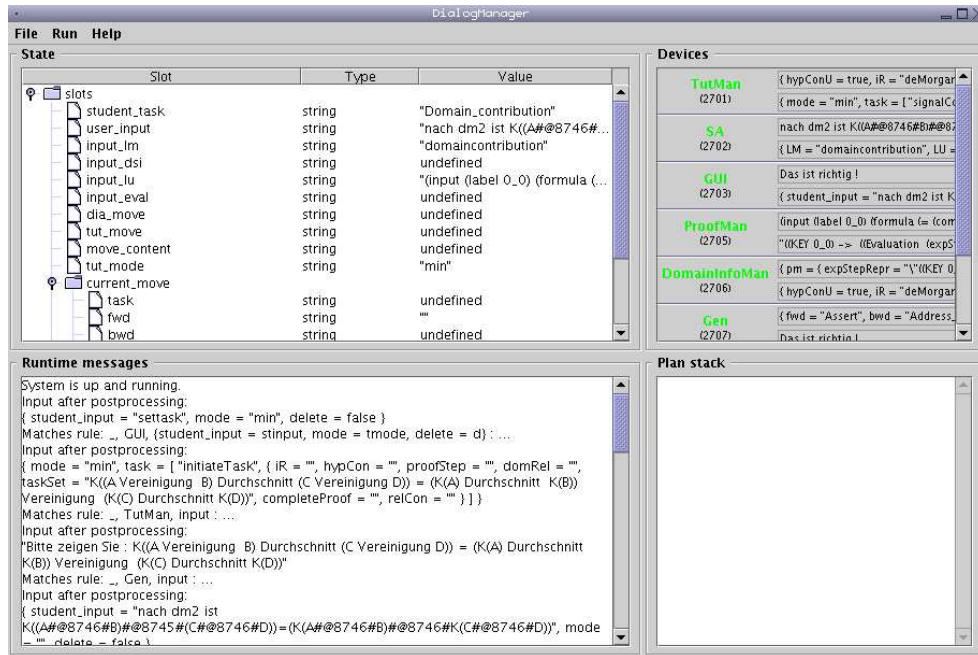
Figure 5.3: The Rubin GUI at the beginning of a demonstrator session.

5.4.2 Rubin's Graphical User Interface

Rubin's graphical user interface shows the details of all communication to and from the Rubin server and the current values in the information state. An example is shown in Figure 5.3. The current values of the IS slots are displayed in the upper left window, and in this window it is possible to alter the values of slots in-place at runtime. This is useful for debugging and testing.

The bottom left area is the server output for each input rule that fires. For each rule execution it shows the input and the header of the rule which fired. In the top right window is a list of the devices that are connected and their ports. For each device the GUI displays the most recent input and output. The current plan stack appears in the lower right area.

5.4.3 Connecting a Module

Rubin offers a simple way to connect external modules to the dialogue manager. It provides the Java abstract class `Client`, from which wrapper classes for each module can be derived, and this wrapper acts as the link between Rubin and the module itself. The wrapper must implement the callback `output(Value v)`, which receives data from the Rubin server, and it sends data back to Rubin with the function `send(Value v)`, as illustrated in Figure 5.4. Both of these functions accept only `Value` objects, which is the internal data type used in communication with the Rubin server and in the dialogue model. Communication is implemented via an XML protocol over a TCP/IP socket connection. Since the wrapper
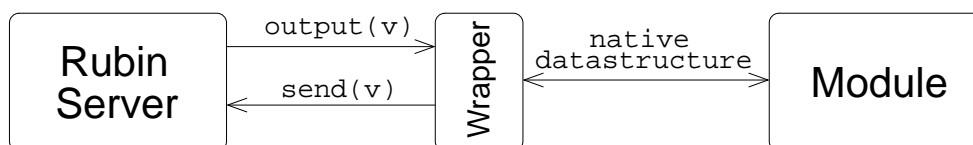
Figure 5.4: Wrapper communication between Rubin and a module.

is written in Java (as opposed to the Rubin–internal syntax of the dialogue model), the full expressivity of the language is available in the wrapper class, for instance to use an existing program's interface functions, to connect to another programming language, or to perform operations on the incoming data from Rubin, such as translation into a native data structure, before making a call to the module itself. Modules are assigned a unique name and a communication port in the wrapper, and these must match the name and port specified in the dialogue model.

5.5   The Dialogue Model

In this section we present the dialogue model for the DIALOG demonstrator. It contains the specification of the information state slots, the device connections, the rules to capture input from modules, and plans for unparsable input. The slots that make up the IS, along with a description of their function and an example of their values, are listed in Table 5.1.

The modules described in Section 5.3 are assigned port numbers for socket communication:

```
TutMan          : 2701;
SA              : 2702;
GUI             : 2703;
ProofMan        : 2705;
DomainInfoMan   : 2706;
Gen             : 2707;
DMR             : 2710;
```

Since our linguistic analysis is performed entirely by a more advanced input analyser (the analysis of mixed natural language and mathematical formulas is one of the core issues of the DIALOG project), the dialogue model does not use the grammar functionality provided by Rubin.

A support function is used to implement the choice of dialogue move for the system (this is a simulated module in the demonstrator). It is a function which takes as input the dialogue move corresponding to the student utterance, specified by its 6 dimensions, and tries to match it against a list of hard-coded dialogue moves. For each possible student dialogue move it returns the dialogue move representing the appropriate system response.

Slot Name	Type	Description and example
student_task	string	The task-level content of the student's last dialogue move. "Domain_contribution"
user_input	string	The user's last utterance. "A und B müssen disjunkt sein."
input_lm	string	Linguistic meaning of the utterance, from input analyser "domaincontribution"
input_lu	string	Underspecified representation of the user's proof step (input (label 1_1) (formula . . .) (see Section 5.3.2)
tut_mode	string	The current tutorial mode "did", "soc" or "min"
current_move	struct	The six dimensions of the dialogue move just performed by the user. {fwd = "Assert", bwd = "Address_statement", commm = "", task = "Domain_contribution", . . . } (see Section 5.3.3)
complete_proof	string	The complete user proof, output by the proof manager when the proof has been completed. ((KEY 1_1) → ((Evaluation (expStepRepr (label 1_1) (formula . . .) (see Section 5.3.4)
deleting	bool	A flag which is set to true when the latest user/system turn is to be undone. (true/false)

Table 5.1: The information state slots in the dialogue model.

This move is underspecified in the sense that the pedagogical knowledge of the tutorial manager has not yet been added in the task dimension.

Plans are used to handle unparsable input, since in this case no mathematical or pedagogical knowledge is required by the dialogue manager, and the corresponding modules therefore do not need to be called. When the dialogue manager receives input from the input analyser stating that the user's utterance was uninterpretable, the dialogue manager sends the generation module a ready–made dialogue move which has in its backward dimension an encoding of why the utterance was not parsed (e.g. due to a parenthesis mismatch). This information can be used in the verbalisation of the move in order to tell the user what was wrong with the input, and to help them correct their error.

5.5.1 Input Rules

The rules section of the dialogue contains the following rules (only the rule headers are listed):

```
_, "Gen", input: {...}
```

```
_, "GUI", { student_input = stinput,
            mode = tmode,
            delete = d} : {...}
_, "SA", { LM = typeof_lm, LU = input} : {...}
_, "DMR", input : {...}
_, "ProofMan", input : {...}
_, "DomainInfoMan", input : {...}
_, "TutMan", input : {...}
```

For each module connected to the dialogue manager there is a rule to capture its input to the Rubin server. None place any constraints on values in the information state. Where the input needs to be analysed to decide on what action the dialogue manager takes, a pattern is used to bind elements of the input to specific variable names.

The input rules in the dialogue model are the concrete realisation of the communication function of the dialogue manager. Because an input rule is specified for each device, the dialogue manager can accept data from each device at any time. In each rule there is a call to the output function, which passes data to another device. In this way, the dialogue manager uses its input rules to create an input/output framework, in which each rule stores its input in the IS, and based on conditional tests, sends output to another module.

```
1.   _, "GUI", { student_input = stinput, mode = tmode, delete = d} : {
2.       slots.user_input = stinput;
3.       //check if the tutorial task is being set
4.       if (stinput == "settask") {
5.           slots.tut_mode = tmode;
6.           // send complete structure with null values to TutMan
7.           output_struct(TutMan, GUI, {delete = false, mode = tmode, ...});
8.       }else if (d==true) {
9.           //what to do if delete
10.          slots.deleting = true;
11.          output_string(SA, GUI, stinput);
12.      }else {
13.          if (tmode != "")
14.              slots.tut_mode = tmode;
15.          output_string(SA, GUI, stinput);
16.      }
17. }
```

Figure 5.5: The input rule for data received from the GUI.

As an example consider the rule for input from the GUI, shown in Figure 5.5. This rule fires only on input where the object which is received is a structure with the field labels student_input, mode, and delete. This is exactly the structure that the GUI sends each

time the user submits an utterance. In the header of the rule matching takes place on the input pattern, and the values in the structure become bound to the local variables `stinput`, `tmode` and `d`. Line 2 shows access to the IS, where the user utterance (a string) is stored in the IS slot `user_input`. This makes it available to other modules for future computations. In this line, `slots` refers to the structure in the dialogue model which contains each of the IS slots.

The next step in this rule is to determine if the user has just started a new dialogue with the system. A new dialogue is started in the demonstration system simply by setting the tutorial mode. In this situation, the token "settask" is sent as the user utterance (even though the user did not really say this), and control switches directly to the tutorial manager to set the task (Line 7). The tutorial manager receives a structure which is empty except for the tutorial mode. In this way the tutorial manager knows that a dialogue is being initialised and in what tutorial mode.

The if-clause in Line 8 tests if the user has decided to delete a move. In this case, the flag `deleting` in the IS is set to true, and control passed to the input analyser (with the device name "SA") by sending it the user utterance which is in the variable `stinput`. The final check is in Line 13, where the rule tests if the tutorial mode has changed. In this case the new tutorial mode is simply stored in the IS, and control passes to the input analyser as usual.

The functionality to delete a pair of user/system turns is also implemented in our dialogue model. When the GUI's output contains the flag `delete = true`, then this value is stored in the IS slot `deleting` to be later passed to the tutorial manager. This is necessary to keep the tutorial manager's model up to date, for instance, of which concepts have been given away, or how many hints have been given.

5.5.2   Information Flow

The input rules described in the previous section give rise to a strict flow of information for each system turn. This is illustrated in Figure 5.6. The diagram shows that when the user's utterance contains no domain contribution, that is, when the user makes no statement about the proof itself, the proof manager, domain information manager (PSM[3]) and tutorial manager are not called for the system response. This reflects the fact that when an utterance has no proof relevant content, there is no need to involve the modules which deal with respective domain knowledge. It suffices to create the system response solely based on dialogue level knowledge, which is encoded in the dialogue manager, the input analyser, dialogue move recogniser, and the NL generator.

Each arrow in the information flow diagram is actually a transfer of control facilitated by the dialogue manager's communication function. As mentioned above, each rule in the dialogue model embodies an "input, process data, output" step, and these steps are shown in the diagram as arrows connecting modules. For example, when the dialogue move recogniser outputs data, the dialogue manager tests whether the dialogue move encodes a

---

[3]Proof Step Manager, a previous name for the domain information manager.
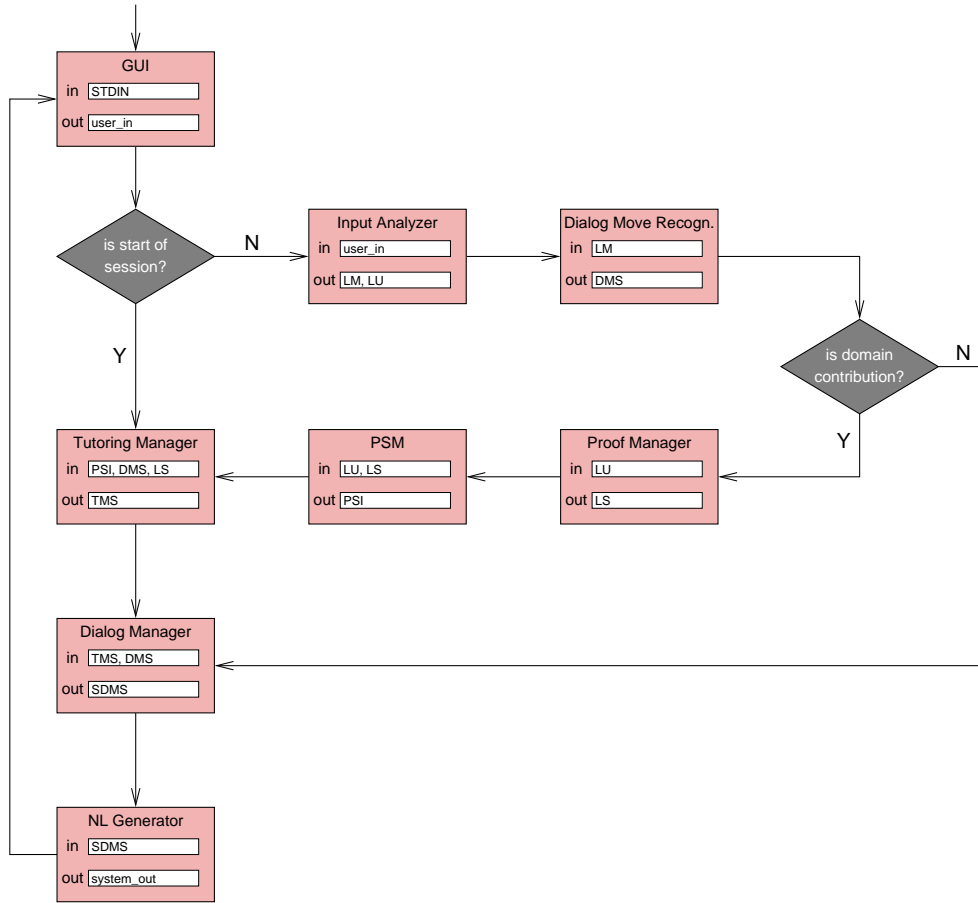
Figure 5.6: Information flow in the Dialog demonstrator for a single system turn.

domain contribution, and based on this, passes control to either the proof manager or to itself.

5.6 Discussion

5.6.1 Module Simulation

A number of modules in the demonstrator which were not yet fully implemented had to be simulated. The natural language generation module had to be simulated because the foreseen generation system, *P.rex*, has not yet been adapted for the Dialog system, or for German language output. The generation module uses a "canned–text" style. It contains a mapping of dialogue moves to strings, and given a set of dialogue moves, returns the corresponding utterance(s).

The domain information manager receives the linguistic meaning and the user proof

step from the input analyser, as well as the evaluation of the proof step from the proof manager. It matches these together against hard–coded lists of input, and outputs the assigned values for the specification of the proof step information of the user's utterance.

The dialogue move recogniser receives the linguistic meaning from the input analyser and returns the dialogue move that corresponds to that utterance by matching against 5 possible types of linguistic meaning. These are: domain contribution, request for assistance, and uninterpretable utterance due to bad grammar, parenthesis mismatch, or a word not in the lexicon.

The wrapper communication with the Rubin server facilitated module simulation since the matching algorithms could be implemented directly in the wrapper class. When a module is then later implemented, it is easy to build it in to the system, because the wrapper interface already exists. In this way internal changes in modules are insulated away from the dialogue manager itself, and only the `output(v)` function needs to be reimplemented to interface with the new real module.

5.6.2 Implementation Issues

A difficulty in achieving a stable, running demonstrator program was interfacing between programming languages in order to connect all modules to the dialogue manager. Rubin, and therefore the dialogue manager built on it, is written in Java, which means that any module to be connected as a device must interface with Java. The GUI and the simulated modules are already written in Java, but the input analyser is written in both Java and Perl, and the tutorial manager and proof manager, as well as the mathematical proof assistant ΩMEGA–CORE are written in LISP.

The solution we decided on for communication with the dialogue manager was to use socket communication, in a similar way to the connection between Rubin and its devices. Using sockets a string can be written to a stream in one programming language and then read from the same stream in another language running as a different process. This allows any two languages to exchange data as long as they support streams and sockets. The disadvantage of this solution is that only strings can be passed through a socket. Each piece of data must be first translated into a string by the sender and then parsed by the receiver, adding an extra layer of complexity to the inter–module communication.

The socket connections were prone to random failures, where strings sent into a stream are not received at the other end. When this happens the process of the module involved had to be stopped, and could only be restarted when the operating system had freed the port, which can take up to a few minutes. In practice this forces a system restart, because modules cannot be dynamically added or removed, and leads to system instability.

An implementation issue with the input analyser was the use of OpenCCG in Linux. Development of the input analyser was done in a Microsoft Windows development environment. When we attempted to move the application to Suse GNU/Linux for use with the demonstrator, the use of Java user preferences in the OpenCCG package led to runtime errors in the input analyser. As a consequence of this the input analyser was run separately to the rest of the system for demonstration purposes.

5.6.3   Advantages using Rubin

Since Rubin is written in Java, it is easy to design prototypes for modules, and to connect modules to the dialogue manager. It also runs on any platform on which the Java 2 JVM is installed. Rubin supports rapid prototyping, and makes it possible to quickly set up a basic dialogue manager which contains an information state, dialogue plans, grammar and update rules, and is therefore suited to a system like Dialog which is still at an early development stage.

5.6.4   What have we learned?

Our experience in building the demonstrator has made us aware of a number of "desiderata" for dialogue applications in the area of natural language tutorial dialogue based on the characteristics of the demonstrator:

Dialog differs from other user interfaces to automated theorem provers in that it aims to support flexible natural language input and output to the tutorial system and therefore to the prover. This means that a very tight interplay of NLU and proof management must take place, and this should be supported by the dialogue manager. The intelligence of the Dialog system is distributed between its subsystems, and thus there must be sophisticated communication between for instance tutorial management, mathematical knowledge management, and natural language generation. We now mention some of the requirements of a dialogue manager for the Dialog project which we have identified and which we consider necessary to support this functionality.

Direct IS access

Modules use the information state to share the information they need to compute results. This can work optimally when each module has read and write access to the IS, and leads to better use of concurrent execution. Modules should also be notified of changes in the IS by triggers attached to slots.

In Rubin this is not possible. That is, there is no way to state, "When slot A is updated, broadcast this event to all devices" so that they can all read the new value. If rules of this type were available, it would obviate the need to pass control to some module at the end of each update rule. Instead, each module could simply watch the IS until all the information it needs is there (i.e. has been updated since the last system utterance), and then execute.

With IS update triggers it would also be possible for modules to use partial information to concurrently compute partial results without having to take full control of system execution. In this situation input rules would simply write values to the IS, and pro-active modules, or "agents" acting on their behalf, would be the main guides of system behaviour.

Runtime flexibility

We see runtime flexibility as an important feature in the future development of the dialogue manager. It would be possible for example to rearrange the ordering of input rules, alter

the IS constraints in the header of a rule, or to register new modules or exchange modules at runtime. This could be a very useful feature for a tutorial dialogue system, as it would then be possible for instance to dynamically add and remove mathematical databases depending on the domain which is being taught.

In Rubin the dialogue model is static. The information state, input rules and all other definitions that make up the dialogue model are specified before runtime, and thereafter cannot be changed. That means it is not possible to make runtime changes to how the dialogue manager behaves.

Another consequence of this relates to dialogue plans. These are also statically defined and cannot be changed at runtime, which is not suitable for the genre of tutorial dialogue. In a dialogue system where information is elicited, such as timetable queries, Rubin's plans allow a degree of adaptivity in that the same information will not be requested twice. This is done by specifying a precondition on the plan such as "IS slot x is undefined", so that if the required information is already in slot x, the plan will not execute. However for tutorial dialogue, this does not offer enough flexibility. In the theorem proving dialogues conducted by the DIALOG demonstrator, IS slots are repeatedly overwritten, so the "undefined" test on slots is not a reliable indication of what the student has said. Since it is impossible to know or predict the whole range of possible student utterances, it is a much more viable approach to begin a tutorial dialogue with a sketchy high-level dialogue plan which places as few as possible constraints on the course of the dialogue [101]. We see the provision for flexible dialogue planning as an important addition to the dialogue manager.

Meta-level control

To achieve a more flexible and adaptable control over rule firing, we believe a meta-level is necessary in which more sophisticated criteria that the ordering of rules could be used to choose the most appropriate next rule. This contrasts with Rubin, where rules fire based solely on their constraints and place in the dialogue model. Such a meta-level would bring a number of benefits to the DIALOG system:

Heuristic control Heuristics for controlling overall system execution could be implemented in the meta-level, for instance, the decision of what module to invoke at what time.

Comparison of IS updates A meta-level could compare different possible IS updates which are triggered by module input. In this situation, an IS update would not simply be made when the first rule fires, rather a number of updates could be computed and compared for appropriateness based on heuristics in the meta-level. The heuristically most appropriate one would then be selected.

Decoupling of IS updates from information flow Since information flow (i.e. what module to call after an IS update) would be determined solely in the meta-level, the definition of IS updates could be made without needing to also define in the rule the effect that the update has on overall system execution. This would greatly simplify
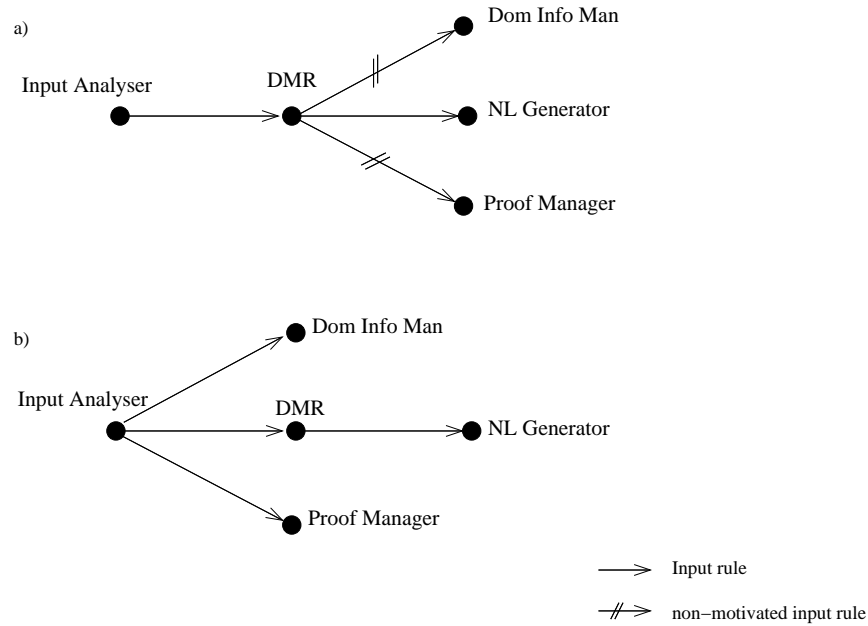
Figure 5.7: A section of the information flow, showing (a) scientifically less well motivated input rules and (b) the equivalent well-motivated input rules.

the design of input rules, because there would be no need to decide on the "next step" of the system within the rule itself.

Flexible flow of control

Since system action is triggered by input from a module which causes an input rule to fire, then if there is no input and no plans on the plan stack, the system stops. This means that in each input rule, the dialogue manager has to pass control to some module which it knows will return some data, and forces a design in which each input rule finishes with a call to output some data to a module so that system execution does not stop. The set of input rules thus forms a chain of invocations, and at all times only one module is executing. All others must wait to be sent information from the dialogue manager before they can execute, even if the information they require has already been assembled within the IS.

We believe a more flexible control of information flow and execution would benefit use of resources and efficiency of computation, as well as helping to facilitate the features mentioned above.

An example of the strict flow of control imposed by the input rules is shown in Figure 5.7. In the figure, an arrow from X to Y represents an input rule which fires on input from module X, and finishes by making a call to module Y, thus forming a link in the chain of module invocations. Part (a) shows the rules as used in the dialogue model for the demonstrator. Here the input from the dialogue move recogniser triggers the invocation of either the domain information manager, the NL generator or the proof manager, depending

on the category of the student's utterance.

For instance, if the utterance was unparsable, the NL generator is called immediately to signal non-understanding. If there is no proof step addressed in the utterance, the proof manager does not need to be called. If the student has made a request for assistance, the domain information manager can be called directly. This conditional branching takes place in the input rule for the dialogue move recogniser. However, this decision is not based on the input from the dialogue move recogniser, but rather on information which came from the input analyser, and could have taken place in the input rule for the input analyser. In this sense the rule for the dialogue move recogniser is not well motivated, and the conditional branching should take place in the rule for the input analyser.

Part (b) shows the scientifically better motivated structure, including rules which it was not possible to use in the demonstrator. For instance, the proof manager could have been called directly after the input analyser. This was not possible because the results of the input analyser have two effects: a call to the proof manager and a call to the dialogue move recogniser. Due to the strict information flow in the dialogue model one of these modules has to "wait" for the other to return before it can be called, leading to the non-motivated rule in (a).

So although the computation that the proof manager carries out is not dependent on the results of the dialogue move recogniser, it is forced to wait until this module finishes its computation. A much better information flow would be achieved by letting these two computations run in parallel rather than in an arbitrarily chosen order.

Overall, this restriction on input rules forces the designer of the dialogue manager to mix information state updates with declarations about the flow of control, since input rules must encode both at the same time. A more intuitive way to define system behaviour would be to declare rules for information state updates and rules for controlling system execution separately, thereby making the definition of both simpler.

5.7 Summary

In this chapter, which ends Part I of the thesis, we have presented the design and implementation of a dialogue manager based on Rubin for the DIALOG demonstrator. As a motivation for the implemented version we began with an overview of the system, and described the basic functionality the dialogue manager should provide as well as its role in the demonstrator. This was followed by an outline of each of the system modules and the computation they perform in turn. The dialogue manager is built on the Rubin platform, which supports the development of ISU based dialogue applications. We introduced Rubin, followed by the definition of the dialogue manager, the so-called dialogue model.

The chapter finished with some desiderata for a dialogue manager that we identified during development. We found that modules did not have direct access to the information state, and that the flow of control was not flexible. Also, there was no meta-level in which we could do reasoning on information state updates.

Part II will present a new dialogue manager based on agent technology which will

support these desiderata in an ISU based dialogue manager.

Part II

Agent-based Dialogue Management

6

The Ω-Ants Suggestion Mechanism

6.1 Introduction

This chapter is the beginning of Part II of the thesis, which presents agent based dialogue management. In Part I we began by introducing dialogue modelling (Chapter 2) and systems which use dialogue management (Chapter 3). In Chapter 4 we gave an outline of the DIALOG project followed in Chapter 5 by an account of the DIALOG demonstrator. This was the first part of our motivation for agent based dialogue management. Our second motivation is the employment of agent technology in Ω-Ants. Agent technology is used to achieve distributed search during proof planning, and to optimise the use of resources by allowing concurrent computation. We propose that these aspects can be carried over to dialogue management and that dialogue management can reap the same benefits from them as theorem proving.

Ω-Ants is a suggestion mechanism which supports interactive theorem proving and proof planning. It is an agent-based system and relies on a hierarchical blackboard architecture. Ω-Ants provides concurrency, runtime flexibility and resource adaptivity in the proof planning process. It has been employed in the ΩMEGA system to automate proof search based on the suggestions of distributed reasoning agents.

In this chapter we begin by introducing proof planning, knowledge based proof planning, and how these are applied in the ΩMEGA system. We then present the architecture of Ω-Ants and show how it uses software agents to achieve distributed proof search. Finally we mention the benefits that this approach brings to theorem proving which we propose to carry over into the field of dialogue management.

6.2   Proof Planning

Proof planning [22] is an extension to tactical theorem proving which employs techniques from AI planning in the field of theorem proving. In tactical theorem proving, the prover abstracts away from the extremely detailed logic level by using tactics. These are called LCF tactics after the proof checker LCF [40]. Tactics are encapsulations of frequently occurring patterns of inference rules combined with commands such as *repeat*, *then* or *else*, called tacticals. Tactics can themselves be combined to form more abstract tactics, until an adequate level of abstraction has been reached.

The central idea of proof planning is to augment tactics with pre- and postconditions, resulting in methods. This transforms tactics into planning operators. If a mathematical problem is considered as a planning problem and the methods as planning operators, then traditional AI planning techniques can be used to build a proof plan. The process of choosing the tactic to be applied in the current proof state must no longer be done manually by the user, as in tactical theorem proving, but can be done automatically by a planner.

Proof planning methods (or more specifically, the tactics that they encapsulate) can be applied in either the backward, forward or sideways direction, or to close a subproof. A backward application of a method can reduce a goal to subgoals, introducing new open goals in the proof state. A forward application can derive new facts from known facts (forward planning), enlarging the set of axioms in the proof. A sideways application can be seen as a special case of either a forward or backward application. It is either a forward application in which new subgoals are introduced in addition to the new fact, or a backward application in which some of the premises of the tactic are given, and only the remaining ones need to be added as new subgoals. A method can also be used to close a subproof if all of its arguments are present.

Ωmega [85] is a mathematical assistance system built around knowledge-based proof planning. In Ωmega the current partial proof plan is stored in a 3-dimensional hierarchical Proof Plan Data Structure ($\mathcal{PDS}$) [23] during the planning process. Each time a method is applied by the planner the $\mathcal{PDS}$ is updated to reflect the new open goals or to mark goals as closed which have been closed in a subproof. The data structure contains all levels of the proof; the logic level as well as the more abstract tactical levels. This is shown schematically in Figure 6.1. At the bottom is the logic layer, which uses a higher-order variant of natural deduction [37]. The higher layers represent the composition of inference steps on the logic level into tactics on a more abstract layer. This continues on up into the higher $\mathcal{PDS}$ layers with increasingly abstract tactics. Proof planning is done at the abstract level of tactics.

In contrast to LCF tactics, Ωmega uses so-called *failing tactics*. These are not guaranteed to be correct, and are not immediately expanded when they are applied. In order to check the correctness of the proof that the planner has produced, the tactics have to be expanded down to the logic layer. This is done by including in each tactic an expansion procedure, which defines how the tactic is to be expanded. In this way a more detailed version of the proof can be computed. This may in turn introduce new tactics. When the
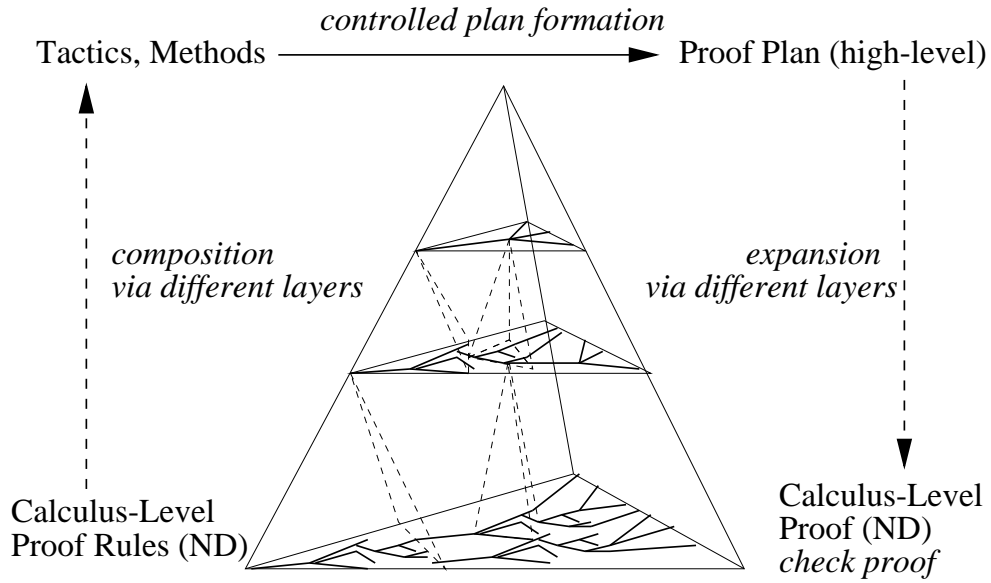
Figure 6.1: The Proof Plan Data Structure.

proof has been completely expanded it can be formally checked on the logic level.

6.3 Knowledge-Based Proof Planning

The ΩMEGA system employs a version of proof planning enriched by mathematical knowledge known as *knowledge-based proof planning* [69]. This is motivated by the fact that mathematicians typically solve problems in a specific area, and in doing so apply techniques which are suitable in that area. In knowledge-based proof planning, this domain specific knowledge is encoded in methods, in control rules, which are general heuristics that guide the planner at choice points, and in external systems. ΩMEGA uses external systems to solve specialised types of subproblems. Types of external reasoners are computer algebra systems containing special computation algorithms, model generators, constraint solvers, and automated theorem provers. An example is OTTER [64], an automated theorem prover for first-order logic.

External reasoners are included in the proof planning process by defining methods which call them. The definition of a method includes application conditions, which are meta-level descriptions that restrict the applicability of a method, and which can consist of arbitrary LISP functions. A method also has outline computations, which allow LISP functions to compute new terms or formulas that the application of a method generates. In both of these places a call can be made to an external reasoner, and thus the method encapsulates the link to the reasoner and allows it to be used in the proof planning process. ΩMEGA does not assume that the external reasoners return correct results, therefore the results of methods which use external systems are expanded down to the logic level in order to be

checked before the proof can be considered correct.

6.4   Ω-Ants: An Agent-based Resource-adaptive System

Ω-Ants [15] is a suggestion mechanism which supports a human user or a proof planner in constructing a proof in ΩMEGA. During an interactive theorem proving session, suggestions for commands and possible instantiations of their arguments based on the current partial proof plan are computed and presented to the user in a graphical interface, ranked according to heuristics. The user can then choose a suggestion and accept, alter or add to the suggested arguments. Ω-Ants can also be integrated into proof planning in ΩMEGA as an algorithm for the planner MULTI [67]. When Ω-Ants is used to support the planner it performs the same task as in the interactive scenario, but it is then the planner which receives the suggestions that Ω-Ants generates. The planner can then choose a suggestion, typically the heuristically highest ranked one. In both the interactive and the automated theorem proving scenario the chosen suggestion is applied to the current proof state stored in the $\mathcal{PDS}$, and Ω-Ants begins computing new suggestions for the updated $\mathcal{PDS}$.

Ω-Ants uses the notions of a command and its partial argument instantiations. A command represents the application of the inference rule (a basic natural deduction rule of ΩMEGA), tactic (abstract proof step), or method (the proof planning operator) with which it is associated. It therefore also includes calls to external reasoners which are embedded in tactics or methods. In order for a command to be applied to the $\mathcal{PDS}$, some (at least) of its arguments must be instantiated. This leads to the concept of a *partial argument instantiation* (PAI), which is a mapping from a subset of the formal arguments of a command to actual arguments. The actual arguments are formulas or proof lines from the $\mathcal{PDS}$. A suggestion which a user receives is thus a command plus its PAI.

6.4.1   Architecture

We can now examine the architecture of the Ω-Ants system, depicted in Figure 6.2[1]. Ω-Ants is a hierarchical blackboard architecture in which societies of software agents collaborate as the source of information for the blackboards. Information flows from the lower level across the blackboards to the top level, where it is presented to the user.

At the lowest level, societies of argument agents search the $\mathcal{PDS}$. Each society represents a command and each argument agent in that society represents an application direction of that command. Argument agents are specified via argument predicates and functions which model dependencies between the formal arguments of the command. Based on a given PAI, argument agents compute new PAIs for positions in the $\mathcal{PDS}$ where the

---

[1]The architecture is turned on its side: the lower level is on the right hand side of the figure, the upper level on the left.
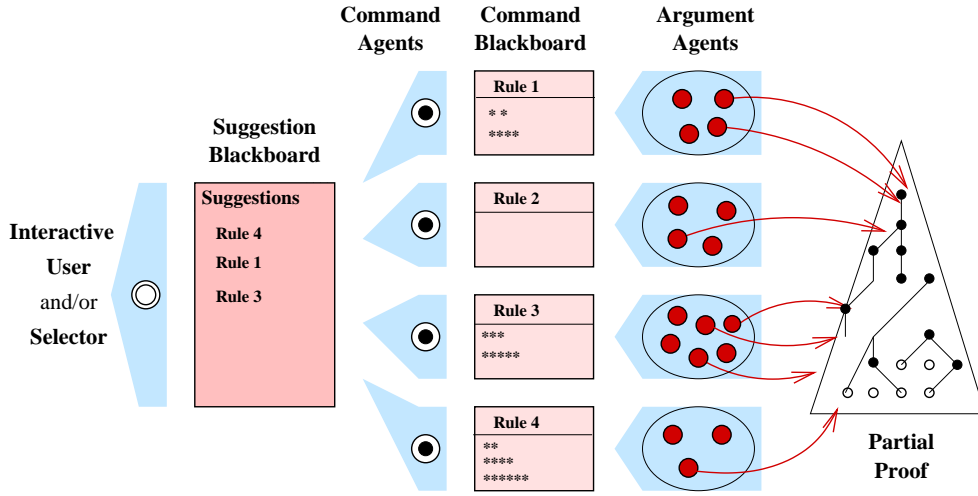
Figure 6.2: The Ω-Ants architecture.

command is applicable. The newly computed PAI is then written to the command blackboard, and the agent restarts its search.

The command blackboard is the next level in the hierarchy. There is one command blackboard for each command and thus for each society of argument agents. Since the argument agents search for new PAIs on the command blackboard and also write their results back there, this is how they exchange results in order to collaboratively compute the most complete PAIs for a command.

Each command blackboard is surveyed by exactly one command agent. Its job is to sort entries on the blackboard according to heuristics. These include preferring the most fully instantiated PAI, or preferring those which have instantiations for a certain subset of the formal arguments. The command agent passes the heuristically preferred PAI on to the suggestion blackboard. It also updates this entry on the suggestion blackboard when it finds a heuristically better PAI on its command blackboard.

The suggestion blackboard gathers the commands applicable in the current $\mathcal{PDS}$ and their respective heuristically preferred PAIs. Each command agent has at most one entry on the suggestion blackboard which it changes accordingly when it finds a new best PAI on its own command blackboard. The suggestion blackboard is monitored in turn by a suggestion agent, the top level of the Ω-Ants hierarchy, which heuristically sorts the entries. For instance, tactics can be preferred over rules because they make larger proof steps, or commands can be preferred which introduce fewer new open subgoals.

The suggestion agent then passes its top entry to the user as the best suggestion, which can be updated as new information filters up to the suggestion blackboard. When the user then executes a command, the Ω-Ants system is reset. This means that the blackboards are cleared, and any agents which are currently computing are interrupted and restarted.

Although Ω-Ants was originally conceived as a suggestion mechanism for interactive proving, it can also be used as an automated theorem prover. A backtracking wrapper is

added on top of the Ω-Ants system which embeds command applications. When either no further suggestions can be found or a timeout is reached Ω-Ants executes the current heuristically preferred suggestion and resets itself. The user can also intervene by explicitly executing a command. This also makes it possible for the user to let Ω-Ants automatically solve some subproblems while the user works interactively on the main problem.

6.4.2 Ω-Ants Argument Agents

In general, a command N in ΩMEGA can be seen as an inference rule of the form

$$\frac{P_1 \ldots P_n}{C_1 \ldots C_m} N \; (T_1 \ldots T_l)$$

where P are premises, C are the conclusions and T are the parameters of the command. Together these make up the formal arguments of the command. An argument agent is defined by three subsets of the formal arguments of the command that it represents:

Goal Set The arguments which the agent is trying to compute.

Dependency Set The arguments which must be present for the agent to carry out its computation.

Exclusion Set The arguments which must be not instantiated.

An example[2] is the inference rule for conjunction introduction, shown here with its corresponding command:

$$\frac{A \quad B}{A \wedge B} \wedge_I \quad \longrightarrow \quad \frac{LConj \quad RConj}{Conj} AndI$$

The set of formal arguments of the command $AndI$ is thus $\{LConj, RConj, Conj\}$. Seen from a proof planning perspective, conjunction introduction can be applied in five different directions: forwards (A and B are given), backwards ($A \wedge B$ is given), two sideways ($A \wedge B$ and either A or B are given) and to close the subproof (all arguments are given). Each of these directions gives rise to a different pattern of goal, dependency and exclusion sets. For instance, the forward application direction would be represented by the goal set $\{Conj\}$, the dependency set $\{LConj, RConj\}$, and an empty exclusion set. Argument agents are further specified by functions and predicates which model dependencies between the formal arguments. In this case, the predicate would ensure that the left conjunct of $Conj$ is equal to $LConj$ and the right conjunct of $Conj$ is equal to $RConj$. Thus we can define a full family of argument agents for a command by giving, for each direction the command can be applied in, the three subsets of the formal arguments along with a function or predicate.

When an argument agent has been defined and its process started it can start to search for suggestions. It begins by searching the command blackboard for a PAI which contains at least instantiations for the arguments in the agent's dependency set, and has

---

[2]from [88]

no instantiations for its goal or exclusion sets. When it finds such a PAI, it marks it as read so that it does not try to complete it again at a later time. The agent then attempts to complete the PAI by searching in the $\mathcal{PDS}$ for an instantiation of the command arguments in its goal set. These must also satisfy the agent's argument function or predicate. When it finds such an instantiation it creates a new PAI which contains exactly the instantiations that the old PAI contained, extended with those that the agent just computed. The new PAI is then written back to the command blackboard. The agent can then return to searching the command blackboard for PAIs it could complete. This type of search, where PAIs are only added and never deleted from the blackboard, means there is no need for concurrent access control on the blackboards.

In the same way as argument agents can be defined for ΩMEGA commands representing inference rules, they can also be defined for commands which call external systems. An example is the first-order prover OTTER, which is integrated into the proof search in ΩMEGA by the command

$$\frac{Premlist}{Conc} Otter$$

Premlist is the list of premises and *Conc* is the conclusion that are sent to OTTER. OTTER attempts to derive the conclusion from the premises, and if it can, sends back the proof object. Thus a command which calls an external reasoner can be treated just like an ordinary command.

By defining argument agents as described here a society of argument agents can compute all possible instantiations for the command which it represents. This is done in a concurrent and distributed fashion. After a sufficient length of time, the command blackboard will contain the optimal instantiations for the command. This is the result of the collaboration of the argument agents which repeatedly read and complete each others PAIs. In doing so argument agents run as separate processes, continually searching for an instantiation which satisfies their goal, dependency and exclusion sets.

6.5   Benefits of Ω-Ants

By employing agent technology and a hierarchical design Ω-Ants brings a number of benefits to both interactive and automated theorem proving. The agent approach brings with it concurrency, which is important for the integration of external reasoners. It also facilitates resource adaptiveness and runtime flexibility since agents can be controlled at the process level. The distributed nature of Ω-Ants makes its proof search robust.

Concurrency   Ω-Ants agents run as independent processes, which means that many agents can concurrently compute suggestions, thus maximising use of system resources. This also leads to an "anytime" character; the system always has some suggestion ready, and the quality of suggestions improves continually over time.

A major benefit of concurrency becomes clear when one considers external systems such as other theorem provers or computer algebra systems. These are used by ΩMEGA

as slave systems, and there are corresponding commands in ΩMEGA which make calls to these systems. The Ω-Ants agents which represent these commands can make calls to the external systems in order to compute suggestions. Due to the concurrency of the agents, many external systems can be used simultaneously, speeding up the suggestion process. Also, the user is free to continue the proof planning process as the external systems are working in the background, so a particularly time-intensive computation does not slow down the main proof planning task. This maximises resource efficiency, since the system is never left waiting for the results of an external computation in order to proceed.

Resource-adaptiveness In Ω-Ants the execution of each agent can be monitored and controlled separately by a resource agent. This means that if any agent is using a lot of system resources but is not contributing to the proof, then it can be allocated less resources for future computations. On the other hand, if the suggestions from an agent are chosen and applied more often by the user, then this agent can receive more resources so that its results are presented sooner. In this way Ω-Ants adapts itself to the computational resources available, and to the particular demands of the theorem at hand.

Runtime Flexibility Ω-Ants includes the functionality to define and add agents at runtime. That is, the user can define a new agent or family of agents, and add these dynamically to the system. For example, if a new external prover becomes available, then the user can simply define a new agent to interface with this system, insert this agent into the system, and receive suggestions from this prover right away. Agents can also be removed or redefined at runtime.

Robustness Since Ω-Ants computes many suggestions for a given proof object, the system does not fail if some agent's computation fails, so errors in distributed computations to not harm the overall mechanism.

6.6 Summary

In this chapter we have seen knowledge-based proof planning in ΩMEGA, and based on this we have presented the design of the Ω-Ants system. We have seen how Ω-Ants uses its agent-based hierarchical architecture to support interactive proof search, and we have seen in detail how argument agents are defined and how they collaborate to generate command suggestions. Finally we have shown the benefits that this brings, in particular with regard to the integration of external systems into the proof search. It is this agent philosophy and the benefits associated with it which we would like introduce into the agent based dialogue manager. We will see now in Chapter 7 how this technology is employed and the effect it has for dialogue management.

7

The Agent-based Dialogue Management Platform

7.1   Introduction

In Chapter 5 we presented the dialogue manager of the DIALOG demonstrator system, and based on our experiences with it, identified some features which we consider desirable for a new dialogue manager in the DIALOG project scenario. In this chapter we propose a solution [19] for developing dialogue management applications which will support these missing features. We present the Agent-based Dialogue Management Platform, ADMP. ADMP will provide the functionality to define a dialogue manager that uses the information state update (ISU) based approach to dialogue management, as introduced in Chapter 3. It will provide for a centrally maintained information state with freely definable slots, a set of update rules which can fire based on values in it, and a hierarchical architecture which facilitates the integration of dialogue control heuristics.

ADMP will borrow from the agent-based technology employed in the Ω-Ants system shown in Chapter 6. This will enable us to build a dialogue manager which has all the functionality of an ISU based manager, but coupled with the benefits of agent-based design from Ω-Ants. Each update rule in the system will be represented by a software agent in the spirit of Ω-Ants. These agents will intermittently check if the preconditions of the rule they represent hold, and if so, will write its effects to an update blackboard. The update blackboard in turn is monitored by an update agent, which applies one or more of the proposed updates to the information state. This completes a single transition from one information state to the next.

This chapter begins in Section 7.2 with the motivations for building ADMP, which come from both the DIALOG demonstrator and from Ω-Ants. We then present the architecture

of the system in Section 7.3, including a formalisation of each component. In Section 7.4 we show how to define an information state and update rules in Admp in order to create a dialogue manager.

7.2 Motivation

In this section we give the motivations for the development of Admp. The first is our experience with the Dialog demonstrator, the second is the agent technology from Ω-Ants.

7.2.1 From the Dialog Demonstrator

Our motivation for Admp is based in part on our experiences with the Dialog demonstrator, as described in Chapter 5. We identified four features of a dialogue manager that we would consider useful for Dialog.

The first is the issue of access to the information state by modules. In the demonstrator, modules could not see changes in the information state. Rather they were dependent on being sent a message by the dialogue manager to trigger a computation. We would like modules to be able to access the information state directly, thus giving them an element of autonomy.

The second feature is the introduction of a meta-level to control rule firing. The demonstrator used a model in which input rules fired based solely on constraints and first-come-first-served rule choice. We would like to use a meta-level to allow us to reason about rules rather than simply letting them fire.

The third feature relates to the flow of control in the demonstrator system. The system was restricted to a sequential flow of information between modules and the dialogue manager. We would like to loosen this by making information flow more flexible, that is, modules must not necessarily wait for others to finish computations which their own computations do not depend on. This will also facilitate concurrency of computation.

Finally, we did not have the ability to change the behaviour of the dialogue manager at runtime. We see this kind of flexibility as beneficial for Dialog. Admp will be able to support each of these desired functions.

7.2.2 From Ω-Ants

The other motivation for Admp is use of agent technology in Ω-Ants, as we have seen in Chapter 6. By taking advantage of the characteristics of Ω-Ants, we will be able to support the features mentioned above in an information state update based dialogue manager. This is based on the similarities we see between Ω-Ants and the dialogue manager.

There are a number of similarities between the Ω-Ants system and an ISU based dialogue manager. As we saw in Figure 6.2, Ω-Ants has at its core the proof data structure, which is the central data structure of the overall proof construction process in Ωmega. It

encodes the current state of the proof construction process and is altered by proof steps which are taken in this process. It allows for concurrent access by multiple agents. In a similar way an ISU based dialogue manager is centred around an information state, a central store of dialogue-level information. It maintains the state of the dialogue and its participants, and is updated when the state of the dialogue changes. It should also support concurrent access.

Ω-Ants uses agents to concurrently compute instantiations of commands, where each instantiation can be seen as a suggestion of an update which can be made to the proof data structure. Suggestions are compared and filtered according to heuristics until a "best" suggestion is presented to the user. Each argument agent has conditions restricting when it can carry out a computation, and the result of the computation is a partial argument instantiation, which is then passed on to the agent's command blackboard.

In the same way a dialogue manager based on ADMP will use software agents to represent the update rules of the dialogue manager. The update rules will have preconditions defining which information state the rule is applicable in, and effects specifying how the information state update, which is the result of the rule, is computed. These agents will search the information state (IS), compute information state updates, and write them to an update blackboard, as in Ω-Ants.

A very important similarity is the integration of external systems. Ω-Ants agents allow the natural and efficient integration of external reasoning systems into the proof planning process in order to carry out tasks for which ΩMEGA is not specialised. As we have shown in the architecture of the DIALOG system, the dialogue manager must also support the interleaving of a number of modules in order for the DIALOG system to operate. In Ω-Ants this integration is facilitated by defining agents which make the call to the external system. In the dialogue manager we will use a similar solution, in which agents will be used which make calls to the modules of the DIALOG system.

Because of these similarities, we propose that the same techniques that make the Ω-Ants system successful in proof planning with external systems can be applied in ISU based dialogue management in systems which rely heavily on separate modules.

7.3 Architecture

7.3.1 Overall Design

The overall architecture of ADMP is illustrated in Figure 7.1. On the right is the information state, the central data structure of the system. It is made up of slots, each of which stores a value. The slots can be read by software agents[1], known as update rule agents, which represent the update rules of the dialogue manager. Each of the update rules has in its preconditions a subset of the set of slots in the information state. When the agent sees that its preconditions hold, the rule is applicable. The agent then computes the IS update

---

[1]From here on when we refer to agents, we mean the update rule agents of ADMP, unless explicitly stated otherwise.
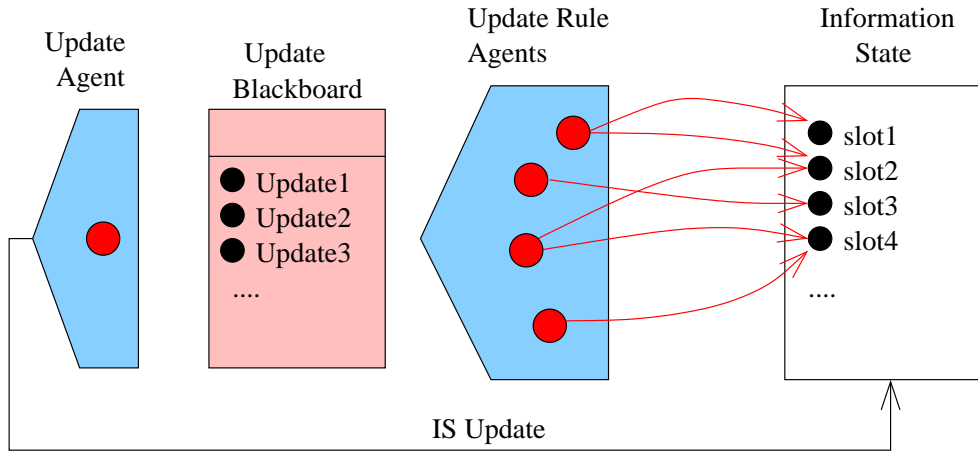
Figure 7.1: The architecture of the dialogue manager.

that is encoded in the rule, and writes this to the update blackboard. This happens in a concurrent fashion, so that many agents may be simultaneously computing results, or may have made calls to external systems. On the far left of the diagram is the update agent, which surveys the update blackboard. After a timeout or some stimulus it chooses the heuristically preferred IS update, executes it, and resets the system. If we recall the architecture of Ω-Ants as shown in Figure 6.2, the similarity in the design of the two systems which was stated as a motivation in the previous section can be recognised.

We now present a detailed formal description of each part of the system. It is important to note here that ADMP is not an agent-based system in the sense of the agent-based theories of dialogue management introduced in Section 3.4.3. Those theories model the actions of dialogue participants as agents, whereas in the case of ADMP, agent-based refers to the technology which forms the dialogue management framework.

7.3.2 The Information State

The information state is the central data structure of the dialogue manager. We see it as a set of typed slots, each with an optional test for legal values of the slot, which enforces typing. Slots and their types are completely freely defined. The values stored in slots can be read and new values can be written to them.

We begin by defining an alphabet $\mathcal{A}$ for slot names, e.g. $\mathcal{A} = \{0, \ldots, 9, a, \ldots, z\}$, and an alphabet $\mathcal{B}$ for the values that can be stored in slots. These alphabets are not restricted here.

Definition 7.3.1 (Alphabet) Let $\mathcal{A}$ be an alphabet. $\mathcal{A}^*$ is the set of words over $\mathcal{A}$ with $\epsilon \notin \mathcal{A}^*$. We define $Identifiers = \mathcal{A}^*$.

Let $\mathcal{B}$ be an alphabet. $\mathcal{B}^*$ is the set of words over $\mathcal{B}$ with $\epsilon \notin \mathcal{B}^*$. We define $Val = \mathcal{B}^*$. $\quad\square$

Identifiers is the language which slot names will be taken from and *Val* is the language in which the objects stored in slots will be expressed.

Typing of slots is provided by defining a boolean function along with each slot. Each time an attempt is made to write a new value to a slot, the function is evaluated for the proposed value. If the function evaluates to true then the proposed value may be written to the slot. Otherwise the change does not take place and the slot retains its existing value. In this way the definition of the test function restricts what values may be written to the slot, thereby enforcing a type. For example, a slot of type string can be declared by defining its test function to only return true for string objects.

Definition 7.3.2 (Test Function on Values) A *boolean test function on values* is a function $f \in \mathcal{BT} := Val \rightarrow \{\top, \bot\}$ ☐

We can now define an information state slot as a tuple consisting of a slot name, taken from the set of identifiers, a type checking function and a value.

Definition 7.3.3 (Information State Slot) Let $s \in Identifiers$, $b \in \mathcal{BT}$ and $v \in Val$. Then the triple (s, b, v) is called an *information state slot*. The set of all information state slots is given by $\mathcal{ISS} := Identifiers \times \mathcal{BT} \times Val$. ☐

An information state slot is denoted by $u = (s, b, v)$, where $s \in Identifiers$ is the name of the slot, $b \in \mathcal{BT}$ is a boolean test or type checking function on the value of the slot, and $v \in Val$ is the value stored in the slot.

In order to retrieve each of the three elements of a slot, we define the following functions to project their values.

Definition 7.3.4 (Projection of slot elements) Given an information state slot $u = (s, b, v)$, we define

$$slotname(u) = s$$
$$slotfunc(u) = b$$
$$slotval(u) = v$$ ☐

We can now define an information state simply as a set of information state slots.

Definition 7.3.5 (Information State) We call $r \subseteq \mathcal{ISS}$ an *information state* if the following conditions hold:

1. $\forall u_1, u_2 \in r$. $slotname(u_1) = slotname(u_2) \Rightarrow$
   $slotfunc(u_1) = slotfunc(u_2) \wedge slotval(u_1) = slotval(u_2)$

2. $r \neq \emptyset$

We define $\mathcal{IS}$ to be the set of all $r \subseteq \mathcal{ISS}$ for which these conditions hold. ☐

$\mathcal{IS}$ is the set of all information states. Each information state in $\mathcal{IS}$ is a right-unique relation, because we require that slot names are unique. It can thus be seen as a mapping from slot identifiers to pairs of functions and values.

It will be useful to be able to talk about the set of all slot names in a given information state. We therefore define the function *slotnames* which returns this set.

Definition 7.3.6 (Slot names) The set of slot names of an information state $r \in \mathcal{IS}$ is given by the function

$$slotnames : \mathcal{IS} \rightarrow \mathcal{P}(Identifiers) \ ,$$
$$slotnames(r) = \{s \in Identifiers \mid \exists \, u \in r \, . \, slotname(u) = s\} \qquad \square$$

7.3.3 Update Rules

In this section we define information state update rules, which represent transitions between information states. An update rule consists of a name, a set of preconditions, a list of sideconditions, and a set of effects. The update rule can fire based on the constraints on the information state which are encoded in its preconditions. It then computes an information state update which can be applied to the information state.

We first describe the notion of an information state update. An information state update is a mapping from slot names to values which represent the changes which are to be made to an information state as a result of an update rule. The domain is the set of slots whose values are to be updated and the image of each slot is the new value that it will have after the update is executed.

Definition 7.3.7 (Information State Update) An *information state update* is a mapping from *Identifiers* to *Val*. The set of all information state updates is denoted by $\mathcal{ISU} := \mathcal{P}(Identifiers \times Val)$. We further define

$$\mathcal{ISU}_\perp := \mathcal{ISU} \cup \{\perp\} \qquad \square$$

$\mu \in \mathcal{ISU}$ is a mapping from symbols to values. An information state update must also take account of the type checking function attached to each slot, and must not attempt to map a slot which does not appear in the current information state. To regulate this we define the concept of *executability* of an information state update in a given information state.

An information state update is executable in a given information state exactly when all of the slots in its domain appear in the information state, and the type checking function for each of those slots returns $\top$ for the given proposed value.

Definition 7.3.8 (Executability) An information state update $\mu \in \mathcal{ISU}$ is *executable* in an information state $r \in \mathcal{IS}$ if

$$\forall (s, v) \in \mu \, . \, \exists \, u \in r \, . \, slotname(u) = s \wedge slotfunc(u)(v) = \top$$

and is denoted by $executable(r, \mu)$. $\qquad \square$

If an information state update is executable in the current information state, we can execute it, thus carrying out the updates that it represents. This is formalised as an *information state update execution.*

Definition 7.3.9 (Information State Update Execution) An *information state update execution* is a function

$$is\_update\_execution \ : \ \mathcal{IS} \times \mathcal{ISU} \to \mathcal{IS}$$

Given an information state $r \in \mathcal{IS}$ and an information state update $\mu \in \mathcal{ISU}$

$$is\_update\_execution(r, \mu) = \begin{cases} execute\_update(r, \mu) & \text{if } executable(r, \mu) = \top \\ r & \text{otherwise} \end{cases}$$

where

$$execute\_update \ : \ \mathcal{IS} \times \mathcal{ISU} \to \mathcal{IS}$$

and[2]

$$execute\_update(r, \mu) = (r \setminus \{(s, b, v) | s \in dom(\mu)\}) \cup r'$$

$$r' = \{(s', b', v')) \mid (s', v') \in \mu \wedge \exists u \in r \,.\, s' = slotname(u) \wedge b' = slotfunc(u)\} \qquad \Box$$

This is the function we use to actually perform information state updates. If the information state update is not executable, the information state r is simply returned. If the information state update is executable, then the slots in r which are in the domain of μ are replaced with slots consisting of the same name, the same type checking function, and the value from the range of μ corresponding to the slot. The set of slot names of the resulting information state is the same as before the update execution. Only those slots which are in the domain of μ have new values, that is, anything which is not explicitly altered remains unchanged.

Now that we have defined information state updates and how they are applied to an information state, we can define the notion of an information state update rule, a description of a transition between information states.

We first define each part of the update rule in turn. The name of the rule is simply an identifier. A precondition is a constraint on the value of a slot in the information state. It is represented by a pair consisting of the name of the slot whose value is constrained and a test function which returns true exactly when the constraint holds. All preconditions must hold with respect to the current information state in order for the rule to fire.

Definition 7.3.10 (Update Rule Precondition) Let $s \in Identifiers$ and $b \in \mathcal{BT}$. The tuple (s, b) is called an *update rule precondition*. The set of all update rule preconditions is denoted by $\mathcal{C} := Identifiers \times \mathcal{BT}$. $\Box$

---

[2]We use $dom(\mu)$ here to mean the set of elements in the first component of every ordered pair in μ

A precondition (s, b) holds when the function b returns $\top$ for the value stored in s.

The sideconditions of a rule add a functional aspect. A sidecondition is a pair consisting of a variable and a function from values to a value. When the sidecondition is executed the result of evaluating the function is stored in the variable. This can be an arbitrary function, which means that an update rule can perform arbitrary computations. This is used for, among other things, making calls to external systems.

Definition 7.3.11 (Update Rule Sidecondition) Let $Var \subseteq Identifiers$ be a set of variables, $v \in Var$ and f be a function. The tuple (v, f) is called an *update rule sidecondition*. The set of all update rule sideconditions is denoted by $\mathcal{D} := Var \times (Val^n \to Val)$ The set of lists of update rule sideconditions $\mathcal{D}_l$ is thus

$$\mathcal{D}_l = \{< d_1, \ldots, d_n > \mid n \in \mathbb{N}, \forall\, 1 \le i \le n \,.\, d_i \in \mathcal{D}\}$$

$n = 0$ gives the empty list of update rule sideconditions. $\qquad\qquad\qquad\qquad\square$

An update rule contains a list of sideconditions which are executed in order. The reason for this ordering is that the variables which are introduced are accessible in the function part of all subsequent sideconditions. As well as this, the names of any slots which appeared in the preconditions of the rule can be used in the function, and these evaluate to the value that the slot had when the rule fired.

An update rule effect is a pair consisting of a slot name and a function over values. The function can include variables which were introduced by sideconditions and slot names which appeared in the preconditions of the rule. When an effect is evaluated the result is the pair containing the slot name from the effect and the result of evaluating the function from the effect.

Definition 7.3.12 (Update Rule Effect) Let $s \in Identifiers$ and f be a function. The tuple (s, f) is called an *update rule effect*. The set of all update rule effects is denoted by $\mathcal{E} := Identifiers \times (Val^n \to Val)$ $\qquad\qquad\qquad\qquad\square$

In this way a set of evaluated effects constitutes an information state update. We can now give the definition of an update rule.

Definition 7.3.13 (Update Rule) An *update rule* ν is given by a tuple

$$\nu \in \mathcal{UR} := \mathcal{U} \times \mathcal{P}(\mathcal{C}) \times \mathcal{D}_l \times \mathcal{P}(\mathcal{E})$$

where $\mathcal{U} = Identifiers$ is the set of names of rules, $\mathcal{C}$ is the set of update rule preconditions, $\mathcal{D}_l$ is the set of lists of update rule sideconditions and $\mathcal{E}$ is the set of update rule effects. $\quad\square$

The general form of an update rule is shown in Figure 7.2, where (s_i, b_i) are preconditions, (s_i, f_i) are effects, and (v_i, f_i) are sideconditions. Given an information state, an update rule computes an information state update based on the constraints which are represented by the preconditions of the rule. We will see in detail how this computation takes place in the next section.

$$\frac{\{(s_1, b_1), \ \ldots \ , (s_j, b_j)\}}{\{(s_1, f_1), \ \ldots \ , (s_l, f_l)\}} n \ < (v_1, f_1), \ \ldots \ , (v_k, f_k) >$$

Figure 7.2: The general form of an update rule.

We constrain the choice of preconditions and effects of an update rule with respect to the current information state by requiring that the slot names which occur in the rule have already been declared in the information state. We also require that the variables introduced by the sideconditions of the rule are unique. If these conditions both hold, then the rule is *well-defined.*

Definition 7.3.14 (Well-definedness) Let $\nu \in \mathcal{UR}$ and $r \in \mathcal{IS}$. The update rule $\nu = (n, c, d, e)$ is *well-defined* w.r.t. the information state r if

$$\{s | (s, b) \in c\} \cup \{s | (s, f) \in e\} \subseteq \mathit{slotnames}(r)$$

and if the sideconditions of the rule constitute a right-unique relation. $\qquad\square$

An update rule can fire when it is applicable with respect to the current information state, that is, when its preconditions are all satisfied by the current information state. We now formalise the notion of applicability of an update rule using the notion of *satisfaction* of update rule preconditions.

Definition 7.3.15 (Satisfaction)

- An information state $r \in \mathcal{IS}$ *satisfies* an update rule precondition $c \in \mathcal{C}$, $c = (s, b)$ if

$$\exists u \in r \, . \, \mathit{slotname}(u) = s \wedge b(\mathit{value}(u)) = \top$$

  This is denoted by $\mathit{satisfies}(r, c)$.

- An information state r *satisfies* a set of preconditions $\mathcal{C}' \subseteq \mathcal{C}$ if

$$\forall c \in \mathcal{C}' \, . \, \mathit{satisfies}(r, c)$$

  This is denoted by $\mathit{satisfies}(r, \mathcal{C}')$. $\qquad\square$

We allow overloading of the *satisfies* predicate where the instance of usage is clear in the context. According to the definition, a precondition is satisfied when the test function returns $\top$ for the value currently stored in the slot in question, and a set of preconditions is satisfied when all preconditions in the set are satisfied. We extend this concept to define *applicability* of an update rule.

Definition 7.3.16 (Applicability) Let $\nu \in \mathcal{UR}$, $\nu = (n, c, d, e)$. ν is *applicable* in an information state $r \in \mathcal{IS}$ if $\mathit{satisfies}(r, c)$ holds. This is denoted by $\mathit{applicable}(r, \nu)$. $\qquad\square$

We can now consider the complete execution of an update rule $\nu = (n, c, d, e)$ when it fires. A rule can execute when it is applicable in the current information state. The first step of the execution is to evaluate the sideconditions of the rule in the order that they appear in the list d. This ordering is required since the expressions can also reference the variables which were bound by sideconditions which appeared earlier in the list d. That is, if $i < j$ for two sideconditions $(v_i, f_i), (v_j, f_j) \in d$, then the variable v_i can appear in f_j, and evaluates to the result of evaluating f_i.

For each sidecondition (v, f) in the set d, the expression f is evaluated and stored in the variable v. In addition to globally and internally declared variables, the expression f can reference the slot names which appear in the preconditions of the rule. These then evaluate to the value that the slot contained when the precondition was satisfied.

The second step in the execution of the rule is to compute the information state update. This is done by evaluating each expression in the set e of effects. In evaluating the effects, the variables which were bound by the sideconditions are available. In this way the results of computations carried out by the rule can be used to update the information state. The result of the update rule is thus an information state update which maps exactly those slots which appear in the effects of the rule.

We can formalise this by defining a function $ur\_execution$ which is called to execute an information state update in a given information state.

Definition 7.3.17 (Update Rule Execution) An *update rule execution* is a function

$$ur\_execution \ : \ \mathcal{IS} \times \mathcal{UR} \to \mathcal{ISU}_\perp$$

Given an information state r and an update rule $\nu = (n, c, d, e)$

$$update\_rule\_execution(r, \nu) = \begin{cases} execute\_rule(r, \nu) & \text{if } applicable(r, \nu) = \top \\ \perp & \text{otherwise} \end{cases}$$

where

$$execute\_rule \ : \ \mathcal{IS} \times \mathcal{UR} \to \mathcal{ISU}$$

and

$$\begin{aligned} execute\_rule \ = \ & \text{for all } (v, f) \text{in } d \\ & \quad \text{let } v := \textbf{evaluate}(f) \\ & \quad \{(s, \textbf{evaluate}(f)) \mid (s_e, f_e) \in e\} \end{aligned}$$

$\square$

Update Rule Agents

Each update rule is represented by an update rule agent which continually checks if the rule is applicable, and if so executes it, thus computing an information state update. In
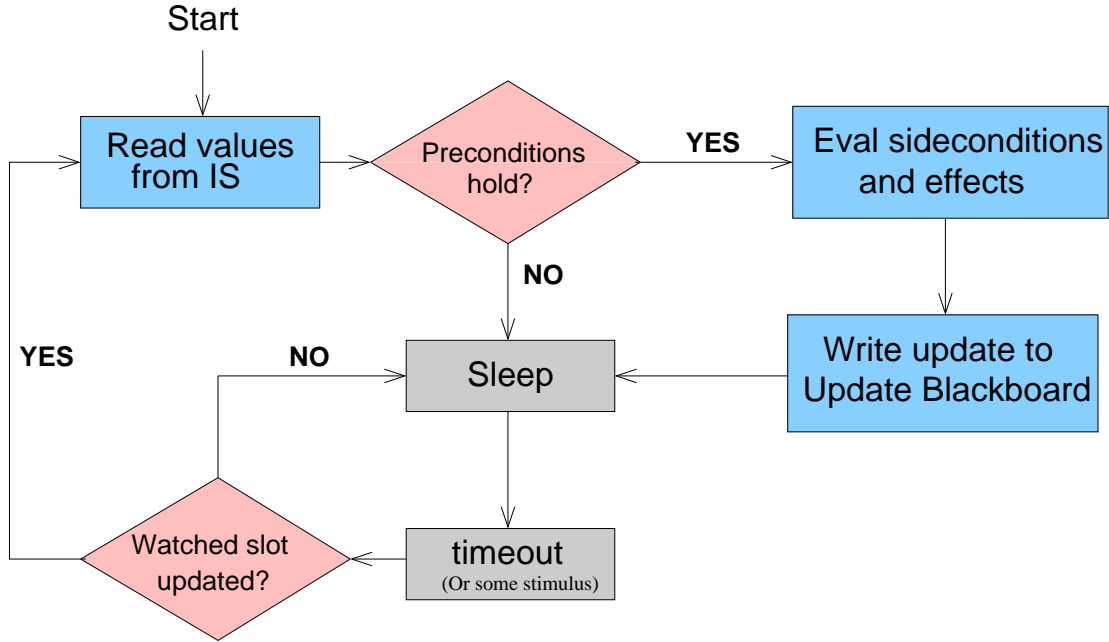
Figure 7.3: The execution loop of an update rule agent.

order to make the repeated checking of the values in information state slots more efficient, we use timestamping to represent when a slot was last updated. Each slot has a timestamp which is set to 0 when the slot is created. Each time a slot is written, its timestamp is incremented by 1. This allows agents which read the slot to determine whether the value contained in the slot has changed since they observed the slot last. If an agent observes that the timestamp of the slot has changed, it knows that an update has been made on the slot. It can use this observation as a trigger for appropriate action in response to a slot update.

The process for the agent representing the update rule $\nu = (n, c, d, e)$ runs as follows: After a certain timeout the agent tests if for any slot s with $(s, b) \in c$, the timestamp of s does not match the timestamp which was saved by the agent the last time it read the value of s. This is the case exactly when the value of the slot has been updated since the agent last read it. The slots it checks are exactly those which appear in the preconditions of the rule, and these are known as the agent's watched slots.

The agent then tests the preconditions of the rule. This means that the preconditions of an update rule only get tested if the update rule agent sees that a value has been updated, which is more efficient than testing all preconditions every time. When a change of a watched slot value is detected, the preconditions are tested, and if they are satisfied, the sideconditions are executed. These have been compiled for each agent into an interpreted function when the update rule was created. Finally the expressions f with $(s, f) \in e$ are evaluated and so the information state update is computed.

Figure 7.3 illustrates the full execution loop of an update rule agent. The initial step

of the update rule agent when its process is started is to test whether its preconditions are satisfied. This involves reading the value of each watched slot, and thus its timestamp, from the information state. If the preconditions are satisfied by the information state, the agent executes the rule and writes the resulting information state update to the update blackboard. It then returns to a sleep state. From the sleep state, the agent intermittently checks whether any of its watched slots have been updated by comparing the timestamp in the IS with the timestamp at which it last read the value of the slot. If a slot has been updated, it reads the values of its watched slots from the IS and again tests whether the preconditions are satisfied. Otherwise it simply returns to the sleep state.

We can also represent the computation carried out by an update rule agent as a lambda term, based on the type of an update rule execution given in Definition (7.3.17). We assume a suitable definition of if-then-else in the lambda calculus.

$$
\begin{aligned}
\lambda r \in \mathcal{IS} \; . \; \lambda \nu \in \mathcal{UR} \; . \quad & \text{if} \; \neg \, applicable(r, \nu) \\
& \text{then} \; \bot \\
& \text{else} \\
& \quad \text{for} \; (v_d, f_d) \text{in} \; d, \; \text{let} \; v_d := \textbf{evaluate}(f_d) \\
& \quad \{(s_e, \textbf{evaluate}(f_e)) \mid (s_e, f_e) \in e\}
\end{aligned}
$$

7.3.4 The Update Blackboard

The update blackboard is the storage space for proposed IS updates.

Definition 7.3.18 (Update Blackboard) An *update blackboard* $w \in \mathcal{P}(\mathcal{ISU})$ is a set of information state updates. □

At any given point in time the update blackboard contains fully instantiated IS updates, namely those which are the effects of the update rules which are applicable at that point, and which have been written to it by the corresponding update rule agent. The update blackboard provides the functionality to write new information state updates onto it (used by update rule agents) and to read its contents (used by the update agent).

7.3.5 The Update Agent

The update agent is the equivalent of the command agent in Ω-Ants which surveys a particular command blackboard. The update agent surveys the update blackboard, comparing and sorting information state updates. After a certain length of time it chooses an update based on heuristics, executes its effects (thus updating the information state) and resets the system. The update agent contains dialogue-level knowledge and heuristics to enable it to choose the "best" update, or union of disjoint updates. We define the function *choose* to formalise this notion.

Definition 7.3.19 (Choice of information state update) The *choice* of an information state update from a set of updates is governed by a function

$$choose \; : \; \mathcal{P}(\mathcal{ISU}) \rightarrow \mathcal{ISU}$$

□

The definition of *choose* defaults to returning the maximal disjoint union of information state updates from $w \in \mathcal{P}(\mathcal{ISU})$. From the definition of information state updates, any union of these will itself by a well-defined update. In this way we can introduce the heuristics to guide the dialogue manager by redefining the function *choose*. Using *choose* we can now define the update agent itself.

Definition 7.3.20 (Update Agent) The *update agent* is a function

$$\tau \; : \; \mathcal{IS} \times \mathcal{P}(\mathcal{ISU}) \to \mathcal{IS}$$

where

$$\tau(r, w) = is\_update\_execution(r, choose(w)) \qquad \square$$

A system reset means that all entries are removed from the update blackboard and all agents are reset, that is, any partial computations are aborted. One of the benefits of the blackboard design is that the update agent, which is where heuristics are built in, does not have to reason about static rule definitions, but rather can reason based on fully instantiated concrete information state updates. This way the update agent has access to the full outcome of any proposed update.

Just as we showed how to represent an update rule as a lambda term in Section 7.3.3, we can represent the update agent in the same way:

$$\lambda r \in \mathcal{IS} \,.\, \lambda w \in \mathcal{P}(\mathcal{ISU}) \,.\, is\_update\_execution(r, choose(w))$$

7.4 Defining a Dialogue Manager

In this section we show how a dialogue manager is defined using ADMP. This is done by defining first an information state and then a set of information state update rules, both in accordance with the specification presented in the last section. We give the syntax of the languages for defining these, and also give an example of each.

7.4.1 Defining the Information State

The information state is declared according to the syntax given in Figure 7.4. As in the definitions above, an information state is specified by a name and a set of slots. A slot is in turn specified by a slot name, an optional documentation string, an optional initial value, an optional test function for type checking, and an optional list of extra arguments to this function. The declaration must obey the following restrictions:

- An information state must have at least one slot.

- *slotname*s must be unique.

```
information-state   ::=   (is~define-is name slots )
name                ::=   :name string
slots               ::=   :slots (slot⁺)
slot                ::=   (slotname [:doc string] [:init value] [:test test-function
                          [:args valuelist]])
valuelist           ::=   (value*)
slotname            ::=   identifier
```

Figure 7.4: Syntax of an information state declaration.

```
(tutorialmode :doc "the current tutorial mode"
              :init "min"
              :test #'(lambda (val list) (find val list :test #'equal))
              :args (("min" "soc" "did")))
```

Figure 7.5: Declaration of the IS slot `tutorialmode`.

- *test-function* is a boolean function which takes the (proposed) value of the slot followed by the arguments specified after the function. The value may be written to the slot if the function applied to the value and the arguments returns ⊤. This defaults to a function which returns ⊤ regardless of input.

- Slots which are not given an initial value default to `nil`.

- A *value* is an element of the set *Val*.

An example of a well defined information state slot is for the tutorial mode, a flag indicating whether we are currently in the didactic, socratic, or minimal feedback mode, shown in Figure 7.5. Four additional values can be specified when a new IS slot is defined. In `:doc` a documentation string for the slot can be given. An initial value can be assigned in `:init`, in this case "min". The `:test` and `:args` values combine to build the test function that implements type checking on the slot. The value of `:test` is a boolean function whose arguments are the proposed value of the slot, followed by the value of `:args` as additional arguments. In the case of the `tutorialmode` slot, the test expands, for a given value v, to

```
(find v ("min" "soc" "did"))
```

This expression, when evaluated, returns true exactly when the proposed value is an element of the list (`"min""soc""did"`), thus ensuring that the slot can only be given one of these values.

7.4.2   Defining the Update Rules

Figure 7.6 gives the syntax of update rule definitions. We require that the name of

$$
\begin{array}{lll}
\textit{update-rule} & ::= & (\texttt{ur\textasciitilde define-update-rule } \textit{name } [\textit{preconds}] \; [\textit{sideconds}] \; [\textit{effects}]) \\
\textit{name} & ::= & \texttt{:name } \textit{string} \\
\textit{preconds} & ::= & \texttt{:preconds } (\textit{precondition}^+) \\
\textit{sideconds} & ::= & \texttt{:sideconds } (\textit{sidecondition}^+) \\
\textit{effects} & ::= & \texttt{:effects } (\textit{effect}^+) \\
\textit{precondition} & ::= & (\textit{slotname } [\texttt{:test } \textit{test-function } [\texttt{:args } \textit{valuelist}]]) \\
\textit{valuelist} & ::= & (\textit{value}^*) \\
\textit{sidecondition} & ::= & (\textit{identifier } \textit{value}) \\
\textit{effect} & ::= & (\textit{slotname } \textit{value})
\end{array}
$$

Figure 7.6: Syntax of update rule declaration.

```
(ur~define-update-rule :name "NL Analyser"
                       :preconds ((utterance :test #'stringp))
                       :sideconds ((result (call-input-analyser utterance))
                                   )
                       :effects ((lm (first result))
                                 (lu (second result)))
                       )
```

Figure 7.7: Declaration of the update rule `NL Analyser`.

the update rule is unique. In Figure 7.7 we give an example of a update rule, showing how it can be used to integrate an external module, namely a natural language analyser: The precondition (`utterance :test #'stringp`) means that the rule fires when the slot `utterance` contains a value which makes `#'stringp` true, i.e. any string. When this is the case, the sidecondition is evaluated. The single sidecondition is a call to the function `call-input-analyser` and takes as input the value of the IS slot `utterance`. In this case it is this sidecondition that forms the interface to the external module. The result is stored in the variable `result`. Next the effects are computed. In this case, the information state update that is computed is of the form $\{(\texttt{lm } x), (\texttt{lu } y)\}$

7.5 Summary

In this chapter we have presented the Agent-based Dialogue Management Platform (ADMP) which is the second and main contribution of this thesis. Its development has a two-fold motivation. The first motivation is the shortcomings we experienced in building the dialogue manager for the dialogue demonstrator. The second is the use of agent-based technology in the Ω-Ants project.

We have presented an architecture which borrows in design from Ω-Ants, and which supports the implementation of information state update based dialogue applications. A

central information state stores the current dialogue context and allows concurrent access by software agents. These software agents represent updates rules which model transitions between information states.

Update rules fire based on constraints on values in the information state, and calculate information state updates. These are collected on the update blackboard which is monitored by the update agent. It chooses which update should be applied and executes it. The final part of this chapter described how a dialogue manager is defined using ADMP. We presented the input language for the information state and for update rules and gave examples of each.

In Section 3.4.4 we saw the components that are necessary in an ISU based dialogue manager, namely an information state, update rules, dialogue moves and an update strategy. ADMP provides each of these, and is therefore suitable for implementing theories of dialogue which use ISU. It also provides the features that were missing from the dialogue manager for the DIALOG demonstrator, such as direct information state access and concurrent module execution.

In the next chapter we will look at these features in more detail and present a worked example which illustrates the functionality of ADMP.

8

Evaluation and Discussion

8.1 Introduction

In the last chapter we gave a formal description of ADMP. We presented the input languages for the information state and the update rules. These are used to define a dialogue manager. Although the scope of this thesis does not permit experimental evaluation, in this chapter we attempt to give an informal evaluation of ADMP.

We begin by presenting an example system to illustrate how a dialogue manager built on ADMP works. The example is a dialogue manager for the DIALOG demonstrator, and we show the information state and the update rules which integrate the system modules. This is followed by a step-through of a single turn of the system showing how the modules interact to compute the system's utterance, with some comments on heuristics. We then briefly discuss how the notion of update rules in ADMP allows a more generic view of the architecture of the demonstrator compared to the system based on Rubin.

After we have presented the example system we compare ADMP to related work in the field. We make some general comments on how ADMP relates to theories of dialogue management from the literature which we introduced in Chapter 3. We first consider the criteria about what role a dialogue manager plays before comparing ADMP directly to three other information state update based toolkits: Rubin, TrindiKit and Dipper.

8.2 The DIALOG Demonstrator using ADMP

In this section we will clarify how ADMP works by giving an instantiation of a dialogue manager built on it. The example we have chosen is the dialogue manager for the DIALOG demonstrator. It will be able to model the same dialogues as the demonstrator. This will

Slot Name	Type
user_utterance	**string**
sys_utterance	**string**
tutorialmode	**string**
LM	**string**
LU	**string**
eval_LU	**string**
proof_step_info	**string**
user_dmove	**dmove**
sys_dmove	**dmove**

Table 8.1: The information state slots of the example system.

allow us to compare the functionality of Rubin, the platform on which the old dialogue manager was built, to that of Admp.

Our aim in presenting this example is to illustrate the functionality that Admp provides, and to show what is required of the developer in building a dialogue manager. It also brings us back to our initial motivation for designing Admp, namely the features which we wanted to introduce in a new dialogue manager after our experience with the demonstrator. We will see that we can reproduce the functionality of the old dialogue manager in Admp, and that the resulting system has a number of benefits.

In order to actually define the behaviour of the dialogue manager, the developer of a dialogue application will define the structure of the information state (IS) and a set of update rules. Here we give the design of the IS and the update rules for the demonstrator system. To make clear how the dialogue manager operates, we follow this with an example of a complete dialogue turn, showing how the update rules fire in order to compute a system utterance. We then mention some issues in relation to the heuristics employed in the example.

8.2.1 Information State

In this example the information state stores the information which must be shared between system modules, or which is collaboratively computed by several modules. We adopt largely the slots from the dialogue manager in the demonstrator. The information state slots are given in Table 8.1.

The slots user_utterance and sys_utterance store the natural language utterances of the student and the system respectively, and are both of type **string**. The current tutorial mode is stored in tutorialmode, and is either "min", "did" or "soc", referring to the minimal feedback, didactic and socratic tutoring styles. LM is a representation in hybrid logic dependency semantics of the linguistic meaning of the utterance, and is computed by the input analyser.

LU and eval_LU are the underspecified and evaluated representations respectively of

the proof content of the student's utterance. The domain information which is addressed in the student's utterance is stored in proof_step_info. Although the content of these three slots has a hierarchical structure as introduced in Section 5.3, the type **string** is used. This is a remnant from the dialogue manager built in Rubin, where a translation to **string** was necessary to send data over the socket connections. An alternative to this representation would be to introduce a type **lu** for LU and eval_LU and a type **proofstepinfo** for proof_step_info, allowing operations and tests on the objects to be done in the dialogue manager.

We assume a type **dmove** whose instances are dialogue moves for the tutorial genre, as described in Section 4.3.2. The objects of this type have operations to retrieve the values of certain dimensions, and we assume a unification operation. The slots user_dmove and sys_dmove both have this type. The slot sys_dmove is an example of a slot whose value is collaboratively computed by several modules. We will see in the section which presents the update rules how this takes place.

8.2.2   Update Rules

In this section we give the update rules of the example system. These model transitions between information states, and form the connection to other modules. The modules which must be integrated are the same as in the old Dialog demonstrator: the proof manager, the tutorial manager, the domain information manager, input analysis, the dialogue move recogniser, and the natural language generator.

Some rules contain only one precondition which does not actually need to impose a constraint on the value of the slot that it reads. Often the only condition for the rule to fire is that the value of the slot has changed since the last turn. We achieve this by simply stating no test function in the precondition of the rule. Two aspects of Admp facilitate this. The first is the fact that a precondition without an explicit test function is interpreted as having the default test function $\top$, a function which returns true regardless of the value of the slot. The second is the notion of timestamping. An update rule agent only reads the value of a slot when it sees that the timestamp has been changed. This means that in a rule with a single precondition, the precondition can only be satisfied when the slot that it constrains has been updated.

These two features combine to allow us to write rules which fire when a single slot is updated simply by stating that slot as a precondition. This is used for example in rule A below, which integrates the input analyser. We now present each update rule in the system in turn. The update rules are given in the form introduced in Figure 7.2 on page 76, and to save space if necessary we will write the sideconditions below the effects of the rule. We will use a more intuitive notation for preconditions rather than the formal declaration syntax given in Section 7.3.3, for instance **has_task(sys_dmove)** to test if the user's dialogue move has a value in the task dimension.

The first rule connects the input analysis module to the system.

$$\frac{\textbf{user\_input}}{\{(\textbf{LM (first result)}),(\textbf{LU (second result)})\}}\,input\text{-}analyser$$
$$< (\textbf{result (call-input-analyser user\_input)}) > \qquad (A)$$

It is an example of a rule which fires based solely on the update of a single slot, here user_input. When this is the case, the function **(call-input-analyser user_input)** in the sidecondition is called, and its return value is stored in **result**. Finally the effects are computed. The input analyser returns two values, the linguistic meaning of the utterance, and a representation of the proof content of the utterance. These are both returned in **result**, and in the effects of the rule they are separated out again so that they can be stored in their respective information state slots. Thus the IS update which is passed to the update blackboard has the form $\{(LM, utt\_lm), (LU, utt\_lu)\}$ where utt_lm and utt_lu are the two components of the input analyser's output.

The next rule integrates the dialogue move recogniser.

$$\frac{\textbf{LM}}{\{(\textbf{user\_dmove result})\}}\,dmr\;<(\textbf{result (call-dmr LM)}) > \qquad (B)$$

Its task is to compute the dialogue move which represents the function of the student's utterance. It can fire when the slot LM has been updated, that is, when the input analyser has stored the linguistic meaning of the utterance in the information state. The effect that this rule computes updates the value of the slot user_dmove.

The following rule represents the link to the proof manager.

$$\frac{\textbf{LU}}{\{(\textbf{evaluated\_LU result})\}}\,proof\text{-}manager\;<(\textbf{result (call-proofman LU)}) > \qquad (C)$$

When the slot LU has been updated, this means that the input analyser has finished its analysis of the student's utterance, and the proof manager should attempt to evaluate the proof content of the utterance. The function **(call-proofman LU)** calls the proof manager, and the result is stored in the information state in the slot evaluated_LU.

After the analysis of the proof step content of the utterance, the system uses domain processing to determine what mathematical concepts were addressed in the step. This computation is carried out by the domain information manager, and is accessed by the update rule:

$$\frac{\textbf{LU}, \textbf{evaluated\_LU}}{\{(\textbf{proof\_step\_info result})\}}\,dim\;<(\textbf{result (call-dim LU evaluated\_LU)}) > \qquad (D)$$

It is applicable when both the LU and the evaluated_LU have been updated in the information state. The result of calling the domain information manager is stored in the slot proof_step_info.

A second rule is also used to interface with the domain information manager. It is applicable in the case where the student's dialogue move was a request for assistance in the task dimension.

$$\frac{\textbf{task\_of(user\_dmove)}=\textbf{"Request\_assistance"}}{\{(\textbf{proof\_step\_info result})\}} \textit{need-help}$$

$$< \textbf{(result (call-dim "" ""))} >$$

(E)

The domain information manager receives the empty strings as input because the student's utterance had no proof content.

When the utterance that the user performed contributes to the dialogue task, then the dialogue move that represents it contains a value in the task dimension. This means that the tutorial manager should be called to address the task content of the dialogue move. Along with the task dimension of the user's dialogue move, the tutorial manager receives the current tutorial mode and the proof step information found by the domain information manager. As well as computing the task content of the system's dialogue move, the tutorial manager can decide to change the tutorial mode.

$$\frac{\textbf{proof\_step\_info}, \textbf{tutorialmode}, \textbf{has\_task(user\_dmove)}}{\{((\textbf{sys\_dmove set\_task\_dim((first result))}), (\textbf{tutorialmode (second result)}))\}} \textit{tutman}$$

$$< \textbf{(result (call-tutman proof\_step\_info tutorialmode task\_of(user\_dmove)))} >$$

(F)

When the task dimension of the system dialogue move has been filled, then the dialogue move is ready to be verbalised by the natural language generation module.

$$\frac{\textbf{has\_task(sys\_dmove), tutorialmode}}{\{(\textbf{sys\_utterance result})\}} \textit{nlgen}$$

$$< \textbf{(result (call-nlg sys\_dmove tutorialmode))} >$$

(G)

This module can use the current tutorial mode to affect how it verbalises the content of the dialogue move. The result is stored in the slot sys_utterance.

There are two update rules which do not call a module, but rather which operate at the dialogue move level within the dialogue manager. The first determines an appropriate system dialogue move given the student's dialogue move. DMM stands for Dialogue Move Manager, and is realised as a hard-coded function **dmm** within the dialogue manager.

$$\frac{\textbf{has\_domain\_task(user\_dmove)}}{\{(\textbf{sys\_dmove dmm(user\_dmove)})\}} \textit{dmm}$$

(H)

The second rule is applicable when the student's dialogue move signals that the student's utterance could not be analysed completely by the input analyser. This can be due to a grammatical error in the utterance or a missing lexicon entry, and is encoded with a token in the task dimension of the dialogue move of either "unint_snu" (for sentence not understood) or "unint_lex". When a formula contains unmatched parentheses this is

detected by the input analyser, and the resulting task dimension value is "unint_par". The system can then use this value to tell the student what his/her mistake was.

$$\frac{\textbf{task\_of(user\_dmove)}=\textbf{"unint*"}}{\{\textbf{(sys\_dmove build\_dmove(user\_dmove))}\}}\textit{erroneous} \qquad (I)$$

Finally we have two rules which interface with the GUI of the DIALOG system. The first rule is applicable when an utterance has been stored in sys_utterance. This means that it is ready to be presented to the user.

$$\frac{\textbf{sys\_utterance}}{\emptyset}\textit{gui-send} < \textbf{(result (send-to-gui sys\_utterance))} > \qquad (J)$$

The function **send-to-gui** causes the GUI to display the system utterance. It returns no value, and there is no associated information state update which must be performed. This is however only in the current version of the DIALOG system. It would be possible, for example, to explicitly model which dialogue participant has the turn, in which case the effect of performing an utterance would be that the system could decide to give up the turn. Since we do not model turn taking, this rule has no effect on the information state.

The corresponding rule to retrieve the student's utterance from the GUI is:

$$\frac{\emptyset}{\{\textbf{((user\_utterance (first result)), (tutorialmode (second result)))}\}}\textit{gui-rec} \qquad (K)$$

$$< \textbf{(result (get-from-gui))} >$$

In the same way as the performance of an utterance by the system has no effect on the information state, the performance of an utterance by the student places no constraints on the information state. The function **get-from-gui** causes the GUI to wait for input from the student. When the student types an utterance in the GUI, this function returns. The return value is the utterance itself, which is stored in user_utterance, and the tutorial mode, in case the tutorial mode is explicitly changed.

8.2.3   An Example Turn

Now that we have presented the declaration of the information state and the update rules of the dialogue manager, we can examine how it behaves at runtime. To do this, we will present the execution of the system over a single turn, showing at each point which rules are applicable and which fire.

We begin with an information state which contains only the student's utterance and the tutorial mode[1].

Slot Name	Value
user_utterance	nach deMorgan-Regel-2 ist $K((A \cup B) \cap (C \cup D)) = (K(A \cup B) \cup K(C \cup D))$
tutorialmode	min

---

[1]For brevity in this section we do not display the slots which do not yet contain a value.

This information state satisfies exactly one rule: rule A, which represents the input analyser. The rule fires, and makes the call to the input analyser, which computes the linguistic meaning and the proof content of the utterance. The information state update that the rule computes is $\{(LM\ lm_u), (LU\ lu_u)\}$, where we abbreviate the content with lm_u and lu_u[2]. This is the only rule that is satisfied, so no other updates are on the update blackboard. The update agent then executes this update, resulting in the following information state:

Slot Name	Value
user_utterance	nach deMorgan-Regel-2 ist $K((A \cup B)) \ldots$
tutorialmode	min
LM	lm_u
LU	lu_u

Now that LM and LU have been updated there are two update rules which are applicable: B and C. These represent the dialogue move recogniser and the proof manager respectively. These now both fire, and the proof manager and the dialogue move recogniser compute in parallel. B writes the update $\{(user\_dmove\ dmove_u)\}$ to the update blackboard, and C writes $\{(eval\_LU\ elu_u)\}$. The update agent uses the default heuristic of taking the maximal disjoint union of updates, which in this case is $\{(eval\_LU\ elu_u), (user\_dmove\ dmove_u)\}$. When this is executed the resulting information state is:

Slot Name	Value
user_utterance	nach deMorgan-Regel-2 ist $K((A \cup B)) \ldots$
tutorialmode	min
LM	lm_u
LU	lu_u
eval_LU	elu_u
user_dmove	$dmove_u$

The next step of the system again shows a concurrent computation. The information state satisfies the rules D and H. D is satisfied because the slot eval_LU has been updated, and calls the domain information manager. It computes the update $\{(proof\_step\_info\ psi_u)\}$. Rule H, which encapsulates the dialogue move manager, can also fire. It does not call a module, but nonetheless its computation takes place concurrently with the domain information manager. The update it computes is $\{(sys\_dmove\ dmove_s)\}$. Again using the default heuristic, the update agent computes and executes the information state update $\{(proof\_step\_info\ psi_u), (sys\_dmove\ dmove_s)\}$:

---

[2]The actual values of the input and output of the modules are the same as those given in Chapter 5.

Slot Name	Value
user_utterance	nach deMorgan-Regel-2 ist $K((A \cup B) \ldots$
tutorialmode	min
LM	lm_u
LU	lu_u
eval_LU	elu_u
proof_step_info	psi_u
user_dmove	$dmove_u$
sys_dmove	$dmove_s$

The user's utterance was a contribution to the partial proof and thus had a task content. Therefore the information state satisfies rule F, representing the tutorial manager, because the user's dialogue move has a value in the task dimension. The tutorial manager now uses this to calculate the task component of the system's dialogue move. The tutorial manager is called, and the rule writes the update $\{(sys\_dmove\ complete\_dmove_s), (tutorialmode\ tmode)\}$ to the update blackboard, where $complete\_dmove_s$ is now the same dialogue move augmented with a value in the task dimension. This is the only rule that fires, and so this update is made to the information state:

Slot Name	Value
user_utterance	nach deMorgan-Regel-2 ist $K((A \cup B) \ldots$
tutorialmode	min
LM	lm_u
LU	lu_u
eval_LU	elu_u
proof_step_info	psi_u
user_dmove	$dmove_u$
sys_dmove	$complete\_dmove_s$

This information state now satisfies rule G because the system dialogue move has a task value. The rule calls the natural language generator to verbalise the dialogue move, resulting in the update $\{(sys\_utterance$ "Das ist richtig!")$\}$. The final rule execution, rule J, sends the system's utterance to the GUI. It makes no change to the information state, and so the final information state after a complete turn is:

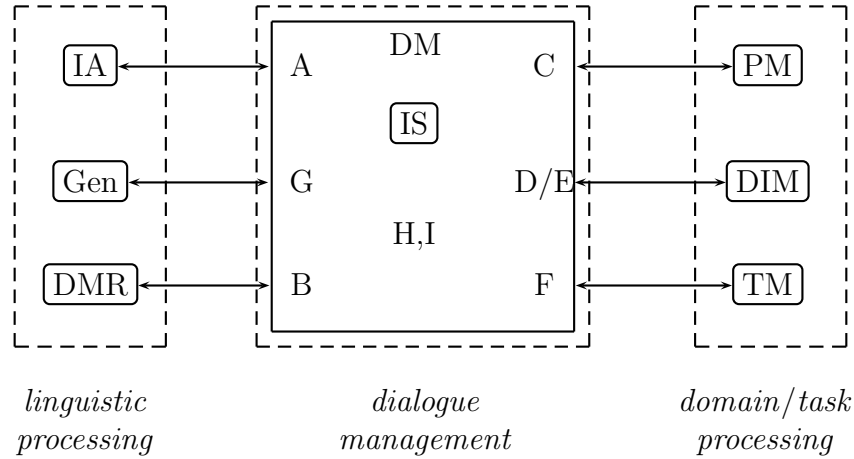| Slot Name	Value
user_utterance	nach deMorgan-Regel-2 ist $K((A \cup B) \ldots$
sys_utterance	Das ist richtig!
tutorialmode	min
LM	lm_u
LU	lu_u
eval_LU	elu_u
proof_step_info	psi_u
user_dmove	$dmove_u$
sys_dmove	$complete\_dmove_s$

Figure 8.1: The architecture of the DIALOG system using ADMP

A Note on Heuristics

The example system we have presented uses the default heuristics from ADMP, that is, the maximal disjoint union of updates is chosen from the update blackboard. In the example system we have presented there are relatively few rules. As a comparison, the GoDis system introduced in Section 3.4.4 has 49 update rules encompassing 7 types [95]. Such an extensive system would offer more scope for guiding the dialogue by way of update heuristics. In this example however the default leads to normal performance.

8.2.4 A New View of the Demonstrator System

In a dialogue manager built on ADMP we use update rules to integrate modules. This means we can visualise the architecture of the system in terms of the links represented by the update rules. This is shown in Figure 8.1[3]. Here each arrow is a link from the dialogue manager to a module, whereby each link is encapsulated by one or more update rules. Letters stand for the update rules as they were introduced in Section 8.2.2. For instance, the input analyser is represented by rule A, and the domain information manager by rules D and E. H and I are "internal" rules in the dialogue manager, i.e. rules which do not interface with a module.

This view of the architecture also allows us to view the system as having subgroups responsible for different tasks. The input analyser, dialogue move recogniser and natural language generator form the linguistic processing part of the system, for both input and output. The interface to this part of the system is encapsulated by the update rules A, B and G. Similarly the task and domain processing part is made up of the proof manager, the tutorial manager, and the domain information manager, and these are interfaced with the rules C, D, E and F.

---

[3]We have deliberately omitted the GUI from this diagram because we want to show only those modules which perform computations in the system.

The definition of the interface to a subsection of the system as a set of update rules is made possible by the update rule concept in ADMP. Each update rule is self-contained and fully specifies the interface to the module it represents. This view of the system also brings us closer to the traditional view of a dialogue system: domain processing and linguistic input and output are treated as separate subsystems which are coordinated by a dialogue management module.

The view of the system shown here would not have been possible with the dialogue manager based on Rubin. If we recall the architecture of the demonstrator shown in Figure 5.1 on page 35, the arrows in that diagram do not represent the input rules of Rubin. This means that the interface to a subset of the modules can not be cleanly specified by a subset of input rules. This shows how ADMP improves the dialogue modelling aspect of the dialogue manager in comparison to Rubin.

8.3 ADMP and Criteria from the Literature

In evaluating ADMP it is important to consider whether it fulfils the requirements for a dialogue manager from the literature on dialogue management. In Chapter 3 we presented the architecture of a dialogue system, followed by the general design of a dialogue manager and the theories of dialogue management which it can implement. Here we look at how ADMP compares to these.

8.3.1 Functions of a Dialogue Manager

A dialogue manager has a number of basic functions which it must fulfil in order to facilitate the dialogue system. We introduced these functions, proposed by McTear, in Section 3.3, and we argue now that ADMP supports each one.

The dialogue manager must be able to maintain a representation of the context of the dialogue. This information is shared between system modules and forms the basis of their computations. ADMP supports this by providing an information state. The structure of the information state is freely definable and thus adaptable to different domains.

Dialogue systems typically use many different modules to perform subtasks such as linguistic analysis or domain processing. In order for each of these to collaborate the dialogue manager must facilitate their communication with one another. In ADMP this feature is achieved by using software agents to represent system modules. The software agents can access the information state, and can make calls to the modules they represent. In this way ADMP can be used to achieve a design like the one shown in Figure 3.1 on page 18.

Finally a dialogue manager should provide the means to control the flow of the dialogue. This includes making appropriate changes to the dialogue context when the state of the dialogue changes, and integrating system modules at the appropriate time. In ADMP this control function is realised in the meta-level. Here it can reason about which update rules should fire, thereby controlling the execution of the whole system.

8.3.2 ADMP in the ISU-based Approach

In Section 3.4.4 we cited the list from Traum and Larsson [95] of the components of an information state update based dialogue manager. ADMP supports the developer of an ISU based application in implementing each of these components, and we now look at each in turn.

Information State   ADMP provides for the declaration of an information state consisting of slots and possibly initial values. The information state can be seen as an attribute-value matrix, and so it is a flexible representation for the dialogue context.

Information State Representations   The information state can only be used when there is some representation for the data which is stored in it. In ADMP we leave the decision of how to represent the content of the information state up to the developer.

The choice of representation is however facilitated by the provision of typing on information state slots. Each slot has an associated type checking function which restricts the values that the slot can contain. The developer can define this function to restrict the values to, say, lists or strings, or more complicated types such as discourse representation structures.

Dialogue Moves   Dialogue moves are an abstraction of the function that utterances have in a dialogue. A theory of dialogue based on the information state update approach will include a theory of dialogue moves, and as such this must be representable in the dialogue manager.

ADMP does not directly support the representation of a set of dialogue moves. We made this decision because of the domain-independence of the system. When a dialogue system is built for a new domain, a set of dialogue moves has to be developed, and we do not want to impose any restriction on how dialogue moves are represented. This is the case for example in the DIALOG project, where the DAMSL schema has been extended to create a taxonomy of tutorial dialogue moves.

Although ADMP does not directly support dialogue moves, the developer can easily introduce them by defining a type **dmove** along with tests for that type, such as unification. It is then possible to write constraints on dialogue moves in the information state, for example:

$$\frac{\{\mathbf{unifies(user\_dmove, somedm)}\}}{<\ is\ update\ >}rulename$$

This states that when the dialogue move stored in **user_dmove** can be unified with the dialogue move **somedm**, then some information state update should be made. In this way dialogue moves can be integrated into constraints on the information state.

Update Rules ADMP provides an input language for writing update rules which consist of preconditions, sideconditions and effects. This allows the developer to specify a transition from one information state to the next. The usual view of an update rule is that it fires and carries out its effects, thereby updating the information state. In ADMP these two events (the rule firing and the effects being carried out) are separated, and we speak of a proposed information state update. However the update rules in ADMP still represent information state transitions.

Update Strategy Traum and Larsson do not specify a general update strategy, that is, the algorithm which chooses the rule which should fire given an information state. The decision of what update strategy to use depends on the domain and the kind of update rules that the system contains.

ADMP also does not enforce a particular update strategy. This part of the dialogue manager is implemented in the choice function which chooses an information state update from the set of updates on the update blackboard.

8.4 ADMP and Related Work

In this section we compare ADMP to related work in the field. We compare it to three dialogue management toolkits which we have presented in this thesis: Rubin, which was introduced in Chapter 5, and TrindiKit and Dipper, which were introduced in Chapter 3.

8.4.1 ADMP and Rubin

Our experience with Rubin was the initial motivation for the development of ADMP. Also, Rubin had already been used in the DIALOG project, the project which forms the framework of this research. Therefore it is useful to consider the similarities of and differences between the two systems. In this section we compare ADMP to Rubin in terms of general functionality, the concept of information state updates, and the desiderata that we identified in Chapter 5.

Functionality

Rubin and ADMP are both platforms for the development of information state update based dialogue applications. As such, both offer the same functionality at the most basic level: an information state along with rules which can alter it. Both also provide the means to integrate modules which run as separate processes.

Rubin provides some features which are not available in ADMP. Rubin includes a GUI with which the information state can be examined and altered in place. This makes it easier to debug the dialogue manager or to test newly connected modules. Rubin also has built in support for connecting modules over an XML protocol, which simplifies inter-module communication and allows distribution of modules over a network. We consider

these two features to be more of a practical nature than a theoretical one. The fact that Admp has no equivalent reduces the ease of use of the system, but not its functionality at a theoretical level. Rubin's information state provides built-in types and record structures. Finally Rubin provides for dialogue plans and for grammars which preprocess input to modules. Although these were not used in the Dialog demonstrator, they are no doubt valuable when building prototype dialogue systems.

Admp has functionalities which exceed those of Rubin. Admp provides for update rules which accurately model information state updates. We show in the next section how these compare to the input rules in Rubin. The other main feature of Admp is the meta-level at which the system can reason about information state updates. This allows the inclusion of heuristics and the flexible overall control of system execution. We consider each of the benefits in turn which Admp has over Rubin, and see that these correspond to the desiderata from Chapter 5

New Update Rule Concept

In comparing Admp with Rubin it is important to highlight the differences between input rules in Rubin and update rules in Admp. These differences have an effect on how modules are called and on how changes are made to the information state.

In Rubin input rules model the transition between the computation carried out by one module and the computation carried out by another module. The beginning of this transition is when the first module sends data to the dialogue manager. This is captured by some input rule. Exactly which rule can fire is governed by the rule constraints on the values in the information state and the content of the data. In the body of the rule changes can be made to both the information state and the current plan stack with the dialogue manager. Finally control is passed to another module so that the system continues to execute. Thus there are always two modules involved in the transition that the input rule represents.

Update rules in Admp, in contrast to the input rules in Rubin, model transitions between information states. The applicability of a rule is determined by constraints on the values in the information state, represented by preconditions. When a rule fires it carries out the computations contained in the sideconditions. This can involve calling another system module. Finally an information state update is computed, which finishes the transition from one information state to the next. There does not have to be any trigger for a rule to fire; the values in the information state are the only constraint.

The update rules in Admp are conceptually simpler and more self-contained than the input rules in Rubin. The reason for this is that Rubin rules must take into account the overall execution of the system because the transition that they model involves two modules. Admp rules only involve one module if at all, and when a module is called by the rule, the computation that the module performs is fully encapsulated in the rule.

Desiderata from the DIALOG Demonstrator

In Section 5.6 we identified a number of desiderata for a dialogue manager which were
motivated by our experience with the DIALOG demonstrator built using Rubin. These
formed the motivation for the development of ADMP. We can now consider how ADMP
supports each of the features that we want to have.

Direct information state access   We found that the modules in the DIALOG demon-
strator had no direct access to the information state, and could not act on changes in the
information state without an explicit stimulus. ADMP, on the other hand, guarantees that
modules which are connected to the dialogue manager always have access to the informa-
tion state, and can react to updates. Since modules are represented by software agents,
they can take autonomous action based solely on the update in the information state and
without the need for an explicit trigger.

Runtime flexibility   In the demonstrator it was not possible to alter the behaviour
of the system at runtime. This meant we could not add or remove modules or change
the declaration of the input rules. While this was not strictly necessary for the dialogue
that was demonstrated, we believe that this feature could be beneficial in the area of
tutorial dialogue. For instance, in a multi-modal tutorial scenario, a graphical interface
for displaying diagrams could be added dynamically when the tutorial reasoning module
decides it would aid understanding.

Changing the system at runtime is supported by ADMP because its agents are modelled
on those used in Ω-Ants. In Ω-Ants agents can be added, removed or suspended at runtime.
In the same way, a new update rule agent can be added to ADMP at runtime, and this new
agent immediately starts monitoring the information state. If the new agent forms the link
to a new module, then this new module is instantly integrated into the system.

Meta-level control   In a dialogue manager based on Rubin control of the execution
of the system is based on the declaration of the input rules. This means that only the
constraints in the header of the rule can control what rule fires. In ADMP the update
blackboard is a meta-level for reasoning on rule execution. Instead of simply allowing each
rule to fire when its constraints hold, the system can reason in the meta-level about which
rule to execute.

The meta-level is a place where heuristics can be introduced into the system. The
heuristics influence how the rule is chosen which should be executed by the system. Heuris-
tics are implemented in the choice function, which determines which update should be
made, given the set of updates on the update blackboard. Another benefit of the meta-level
is that the reasoning that takes place is done on the basis of fully instantiated information
state updates, and not on rule declarations.

Flexible flow of control   In the DIALOG demonstrator the action of the system was
driven by the input rules which linked modules to the dialogue manager. This meant

that it was not possible to separately control the system execution independently of the declaration of the input rules.

We solve this in ADMP by decoupling the declaration of update rules from the execution of the system. An update rule models an information state update, and is constrained only by the information state, not by what rules fire before or after it. There is also no ordering on update rules. The actual execution of the system is guided by the choice function at the meta-level, which decides what update from the update blackboard to perform. Another reason why this decoupling is possible is because ADMP provides for concurrent execution of system modules. Modules can compute in parallel, so there is no need to decide in which order they should be invoked.

8.4.2 TrindiKit and Dipper

In the evaluation of ADMP so far in this chapter we have considered its functionality only in comparison to that of Rubin. However, there are many other dialogue management toolkits and platforms apart from Rubin. Rubin has its particular concept of dialogue modelling, so a comparison with other platforms will allow us to better illustrate the more generic features of ADMP and to make more general comments.

In Chapter 3 we introduced the information state update approach to dialogue management before going on to present two architectures for developing ISU based applications: TrindiKit and Dipper. These are suitable for a comparison to ADMP because both provide a generic framework for ISU based dialogue management and have been successfully used to build concrete systems.

TrindiKit is a fully featured dialogue management toolkit which proposes a general architecture for ISU based systems. When compared with ADMP, one of the main differences is that TrindiKit provides more support to the developer from a software engineering point of view. It specifies formats for defining information states, update rules, and dialogue moves, as well as the algorithms that control system execution. For instance, an input language is provided for defining the update strategy. TrindiKit also facilitates reusability by splitting the system into three layers, only the uppermost of which is domain dependent.

Dipper, on the other hand, is a more stripped-down implementation of the ISU approach. The main similarity it has to ADMP is that Dipper uses software agents to integrate modules into the system. In this regard Dipper has an advantage over ADMP, because it uses the Open Agent Architecture (OAA), a standard framework for inter-agent communication and collaboration. OAA agents represent each module, the dialogue move engine and the dialogue move engine server.

Like TrindiKit, Dipper also provides data types to aid the formalisation of the information state, such as stack, queue, and even discourse representation structure. Dipper uses a GUI to display the current information state and the rules which fire as the dialogue progresses. In contrast to ADMP, Dipper does not have an adaptable update strategy. Instead, update are chosen based simply on rule ordering and rule constraints.

ADMP, TrindiKit and Dipper have many common features. Each supports the ISU approach and provides for an information state, update rules and dialogue moves. Each

also allows concurrent module execution and has a strategy for choosing information state updates. What sets ADMP apart from TrindiKit and Dipper is that it introduces a meta-level into the dialogue system architecture. It is at the meta-level that the update strategy is applied. Dipper has the simplest update strategy: choose the first rule whose constraints are satisfied. TrindiKit allows the developer to implement a more sophisticated update strategy. However both of these approaches are limited to reasoning about rule definitions.

The update strategy in ADMP is implemented in the meta-level. This means that it does not reason about rule definitions, but rather about fully instantiated information state updates. The update agent, which encapsulates the heuristics that affect the update strategy, therefore sees exactly what changes will be made to the information state if a particular update is chosen.

8.5   Summary

This chapter has presented an evaluation and discussion of ADMP. In Section 8.2 we used an example of a concrete system to illustrate how ADMP works. Such a system is defined by an information state and a set of update rules. The example system is a dialogue manager for the DIALOG demonstrator, and allows us to highlight the main features of ADMP. These features are an information state with freely definable slots, update rules which are represented by software agents and which integrate system modules, and a meta-level to heuristically control the execution of information state updates.

The example system shows one of the benefits of ADMP with respect to dialogue modelling. In Section 8.2.4 we grouped the subparts of the DIALOG system which perform linguistic and domain processing, and showed that the interface to these subparts of the system can be specified by sets of update rules. This view of the system is made possible by the design of the update rules in ADMP, which can fully encapsulate the computation of system modules.

We continued in Section 8.3 with an evaluation of ADMP in terms of general descriptions of dialogue systems from the literature. ADMP fulfils the requirements of dialogue management set out in the literature, since it provides for the maintenance of a dialogue context, integration of modules, and overall control of the dialogue flow. Furthermore, a dialogue manager built with ADMP supports or allows the integration of each of the components of an information state update manager as proposed by Traum and Larsson.

In Section 8.4 we compared ADMP to three other toolkits — Rubin, TrindiKit and Dipper — which support information state update based dialogue applications. We concluded that ADMP, as well as each of these toolkits, provides the basic building blocks of an ISU based application, namely an information state and update rules.

However the three toolkits differ in the degree to which they support the software engineering aspect of developing a dialogue manager. TrindiKit for instance provides a layered architecture along with predefined data types which aids reusability. Each of the three also have an interface for debugging and inspection of the information state. Only Dipper uses an agent model for the integration of modules, which utilises the Open Agent

Architecture.

Admp has an advantage over the toolkits we have considered here in that it has a meta-level at which it can reason about information state updates. This is an improvement over the update strategies of the other toolkits. Rubin uses both information state constraints and the module from which the input came to choose an input rule. In Dipper, update rules are chosen based solely on information state constraints. TrindiKit provides a language to define update strategies. However all of these reason only about rules. In Admp, the heuristics in the meta-level reason on fully instantiated information state updates.

In summary, Admp fulfils the general requirements of supporting the development of dialogue applications, and provides the same fundamental functionalities as other ISU based toolkits. What it does not provide is the software engineering aspect; predefined data types or debugging facilities are not available. While this affects the "usability" of Admp, it does not detract from its support for the information state update approach. In fact, Admp adds to this by introducing the meta-level to control system execution.

9

Conclusion and Outlook

Summary of the Thesis

In this thesis we have investigated the application of software agent techniques from inter-active theorem proving to the area of dialogue management. The scenario of the research was the DIALOG project, which is concerned with the modelling and realisation of natural language tutorial dialogue on mathematical proofs. Modelling this type of dialogue puts particular demands on a dialogue manager. For instance, the dialogue manager must main-tain a dialogue context, it must be able to integrate linguistic, task and domain processing, and it must be able to control the dialogue flow.

The result of our research is a platform for building a dialogue manager which uses agent technology and a hierarchical architecture, and which supports the information state update approach to dialogue management.

Motivations

Our motivations for this research came from two directions. The first was our experience in building a dialogue manager for the DIALOG demonstrator, which is documented in Chapter 5. We decided on the information state update approach to dialogue management, and built the dialogue manager using the Rubin platform. However in the course of the development we identified a number of features we wanted to have in an ISU based dialogue manager for the DIALOG project. The integration of these features into a new dialogue manager was one of our main goals.

The second motivation was the use of the software agent philosophy of Ω-Ants, which we treated in Chapter 6. We noted the similarities between the design of Ω-Ants and the design of an ISU based dialogue manager, for instance the use of a central data structure,

operations that update it defined by rules, and the integration of external systems. Ω-Ants also has a hierarchical architecture which allows the integration of heuristics. These are all features which we wanted to carry over into dialogue management.

Contributions of the Thesis

There are two main contributions of this thesis. The first is the design and implementation of the dialogue manager for the DIALOG demonstrator. Building the dialogue manager involved designing an information state that could represent all of the shared information in the system, and declaring rules which on the one hand integrated the system modules, and on the other hand allowed them to interleave their computations.

   The second and most important contribution of the thesis is ADMP, the Agent-based Dialogue Management Platform, which is presented in Chapter 7. Our approach in the development of ADMP was to apply the agent techniques from Ω-Ants to build a platform for dialogue management which provides the complete functionality of an ISU based dialogue manager. We gave a full formal description of the system and a specification of the languages for declaring information states and update rules. In Chapter 8 we illustrated ADMP at hand of an example system and compared it to related work in the field.

Further Work

The results of this thesis suggest two general directions for future work. The first is to consider how ADMP can be improved as a reusable platform for dialogue management.

Software engineering support Established platforms such as TrindiKit have shown the usefulness of features which aid the developer of a dialogue application. These include a GUI to help debug the running system, support for connections to modules, and predefined data types. We hope to add such features to ADMP to improve its ease of use. An example would be adding extensible types such as records or dialogue moves to the information state declaration language.

Dialogue heuristics One thing which is missing from ADMP is a concept of how more complex heuristics should be integrated into the system. We would like to investigate on the one hand, what these heuristics are, and on the other hand, what is required to formalise them. This would involve for instance, designing a declaration language for heuristics on information state updates.

   The second direction of future work is in the context of the DIALOG project, the framework in which ADMP was developed.

Adaptation to developing modules As the DIALOG system develops during the coming project phase, the dialogue manager (more specifically, the dialogue manager built on ADMP) will be adapted to improvements and alterations to other system modules. Because module interfaces are defined in the update rules, adding or altering an interface simply involves rewriting its update rules. This is a ongoing process

which goes hand in hand with the continuing development of each key module in the second phase of the DIALOG project.

Account of barge-in We believe that ADMP and the notion of dialogue modelling that it supports can be utilised to account for dialogue phenomena which are not yet treated by the DIALOG system. An example is barge-in, a phenomenon which can occur in tutorial scenarios. Because of the agent-based design of ADMP, a linguistic analysis module can always be ready to accept input even if other parts of the system are still computing. However it is not yet clear how this would affect the interleaving of linguistic analysis with other system modules.

Outlook

The trend in dialogue systems in the last number of years has been moving towards reusability and domain independence. This is reflected in the availability and utilisation of off-the-shelf components such as speech recognisers, speech generators and planning components. Another indication of this trend is the emergence of generic dialogue management toolkits such as TrindiKit. We believe that the pluggable integration of subsystems in a dialogue system can be achieved using an agent-based paradigm — the Dipper framework has already shown how to facilitate system building using the Open Agent Architecture.

With ADMP we have taken both the agent philosophy and the hierarchical design of Ω-Ants and transferred it to dialogue management. In doing so we have given a framework in which heuristics can be applied to overall dialogue control. Much work has been done on strategies for dialogue control, for example [59], and as this line of research continues it will be important to have dialogue management architectures which support the integration of heuristics in a reusable and generic framework.

Dialogue soc20p

Dialogue soc20p, taken from the WoZ corpus [100].

T1: Bitte zeigen Sie: $A \cap B \in P ((A \cup C) \cap (B \cup C))$!

S1: (`correct`) es gilt: $(A \cup C) \cap (B \cup C) = C \cup (A \cap B)$.

T2: Richtig.

S2: (`correct`) Daraus ergibt sich : $P ((A \cup C) \cap (B \cup C)) = P (C \cup (A \cap B))$.

T3: Ja , das folgt daraus.

S3: (`unknown`) und für die Potenzmenge gilt : $P (C \cup (A \cap B)) = P (C) \cup P (A \cap B)$.

T4: Meinen Sie wirklich : $P (C \cup (A \cap B)) = P (C) \cup P (A \cap B)$?

S4: (`wrong`) ich denke doch : $P (A) \cup P (B) = P (A \cup B)$.

T5: Das ist nicht richtig! Vielleicht sollten Sie noch einmal in Ihrem Begleitmaterial nachsehen.

S5: (`unknown`) entschuldigung, es gilt natürlich : $P (C \cup (A \cap B)) \subseteq P (C) \cup P (A \cap B)$.

T6: Wirklich?

S6: (`correct`) nein doch nicht.. .andersrum.

T7: Das ist endlich richtig!

S7: (`correct`) also gilt: P (C) $\cup$ P (A $\cap$ B) $\subseteq$ P (C $\cup$ (A $\cap$ B)) und A $\cap$ B $\in$ P (A $\cap$ B) woraus der beweis folgt.

T8: Das stimmt. Ich wiederhole noch einmal: Wir wollen beweisen, daß A $\cap$ B $\in$ P ((A $\cup$ C) $\cap$ (B $\cup$ C)) gilt. Wir sehen, daß P ((A $\cup$ C) $\cap$ (B $\cup$ C)) = P ((A $\cap$ B) $\cup$ C) $\supseteq$ P (C) $\cup$ P (A $\cap$ B) $\supseteq$ P (A $\cap$ B). Da aber A $\cap$ B $\in$ P (A $\cap$ B) gilt, ist die Behauptung bewiesen.

B
Dialogue did16k

The dialogue did16k conducted by the DIALOG demonstrator, taken from [100].

∗ Start in Minimal Feedback mode. ∗

T0: Bitte zeigen Sie : K ((A ∪ B) ∩ (C ∪ D)) = (K (A) ∩ K (B)) ∪ (K (C) ∩ K (D)) !

S1: (`correct`) nach deMorgan-Regel-2 ist K ((A ∪ B) ∩ (C ∪ D)) = (K (A ∪ B) ∪ K (C ∪ D))

T1: Das ist richtig !

S2: (`correct`) K (A ∪ B) ist laut DeMorgan-1 K (A) ∩ K (B)

T2: Das stimmt auch !

∗ Undo S2-T2 ∗

S2′: (`wrong`) B enthält kein x ∈ A

T2′: Das ist nicht richtig !

∗ Switch to Socratic mode. ∗

S3′: (`wrong`) A und B müßen disjunkt sein .

T3′: <*hint*>

∗ Undo S3′-T3′ ∗

S3″: (`irrelevant`) A=K(K((A))

T3″: $<hint>$

S4″: (`unparsable`) $<Out\ of\ grammar\ input.>$

T4″: Diese Satz kann nicht interpretiert werden. Bitte formulieren Sie die Eingabe um.

S5″: (`unknown-word`) K (C ∪ D) würde laut DeMorgan-1 K (C) ∩ K (D)

T5″: Das Wort "würde" kann nicht interpretiert werden. Bitte formulieren Sie die Eingabe um.

S6″: (`correct`) K (C ∪ D) ist laut DeMorgan-1 K (C) ∩ K (D)

T6″: Das ist richtig !

∗ Undo S6″-T6″ ∗

S6‴: Bitte hilfe !

T6‴: $<Give\ away\ step.>$

S7‴: (`correct`) Also folgt : K ((A ∪ B) ∩ (C ∪ D)) = (K (A) ∩ K (B)) ∪ (K (C) ∩ K (D)) .

T7⁗: Das stimmt genau . Ich wiederhole noch einmal : Auf die linke Seite der Gleichung kann ich zuerst die zweite und danach die erste de-Morgan-Regel anwenden , so daß sich folgende Argumentationskette ergibt : K ((A ∪ B) ∩ (C ∪ D)) = K (A ∪ B) ∪ K (C ∪ D) = (K (A) ∩ K (B)) ∪ (K (C) ∩ K (D)) .

Bibliography

[1] Vincent Aleven, Kenneth R. Koedinger, and K. Cross. Tutoring answer explanation fosters learning with understanding. In S. P. Lajoie and M. Vivet, editors, *Artificial Intelligence in Education, Open Learning Environments: New Computational Technologies to Support Learning, Exploration, and Collaboration, proceedings of AIED-99*, pages 199–206, Amsterdam, 1999. IOS Press.

[2] James Allen, Donna Byron, Myroslava Dzikovska, George Ferguson, Lucian Galescu, and Amanda Stent. An architecture for a generic dialogue shell. *Journal of Natural Language Engineering, special issue on Best Practices in Spoken Language Dialogue Systems Engineering*, 6(3):1–16, December 2000.

[3] James Allen, Donna Byron, Myroslava Dzikovska, George Ferguson, Lucian Galescu, and Amanda Stent. Towards Conversational Human-Computer Interaction. *AI magazine*, 2001.

[4] James Allen and Mark Core. Draft of DAMSL: Dialogue act markup in several layers. *DRI: Discourse Research Initiative*, University of Pennsylvania, 1997.

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

[6] José Gabriel Amores and José F. Quesada. Dialogue moves in natural command languages. Siridus deliverable D1.1, SRI International, Cambridge, 2000.

[7] Nicholas Asher. *Reference to Abstract objects in Discourse*. Kluwer Academic Publishers, 1993.

[8] Harald Aust, Martin Oerder, Frank Seide, and Volker Steinbiss. The Philips Automatic Train Timetable Information System. In *Speech Communication*, volume 17, pages 249–262, 1995.

[9] John L. Austin. *How to Do Things with Words*. Harvard University Press, 1996.

[10] Serge Autexier, Christoph Benzmüller, Armin Fiedler, Helmut Horacek, and Bao Quoc Vo. Assertion-level proof representation with under-specification. *Electronic Notes in Theoretical Computer Science*, 93:5–23, 2003.

[11] Jason Baldridge and Geert-Jan M. Kruijff. Coupling CCG with Hybrid Logic Dependency Semantics. In *Proceedings of the 40th Annual Meeting of the Association of Computational Linguistics (ACL'02), June 7-12*, University of Pennsylvania, Philadelphia, 2002.

[12] Jason Baldridge and Geert-Jan M. Kruijff. Multi-modal combinatory categorial grammar. In *EACL '03: Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics*, pages 211–218. Association for Computational Linguistics, 2003.

[13] Chris Benzmüller, Armin Fiedler, Malte Gabsdil, Helmut Horacek, Ivana Kruijff-Korbayová, Manfred Pinkal, Jörg Siekmann, Dimitra Tsovaltzi, Bao Quoc Vo, and Magdalena Wolska. Tutorial dialogs on mathematical proofs. In *Proceedings of the IJCAI Workshop on Knowledge Representation and Automated Reasoning for E-Learning Systems*, pages 12–22, Acapulco, 2003.

[14] Christoph Benzmüller, Armin Fiedler, Malte Gabsdil, Helmut Horacek, Ivana Kruijff-Korbayova, Dimitra Tsovaltzi, Bao Quoc Vo, and Magdalena Wolska. Language phenomena in tutorial dialogs on mathematical proofs. In *Proceedings of the 7th Workshop on the semantics and pragmatics of dialogue (DiaBruck)*, Wallerfangen, Germany, 2003.

[15] Christoph Benzmüller and Volker Sorge. OANTS – an open approach at combining interactive and automated theorem proving. In Manfred Kerber and Michael Kohlhase, editors, *Symbolic Computation and Automated Reasoning*, pages 81–97. A.K.Peters, 2000.

[16] Nate Blaylock, James Allen, and George Ferguson. Managing communicative intentions with collaborative problem solving. In Jan van Kuppevelt and Ronnie W. Smith, editors, *Current and New Directions in Discourse and Dialogue*, volume 22 of *Kluwer Series on Text, Speech and Language Technology*, pages 63–84, Kluwer, Dordrecht, 2003.

[17] Johan Bos, Stina Ericsson, Staffan Larsson, Ian Lewin, Peter Ljungloef, and Colin Matheson. Dialogue dynamics in restricted dialogue systems. TRINDI Deliverable D3.2, 2000.

[18] 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.

[19] Mark Buckley. An Agent-based Platform for Dialogue Management. In Judit Gervain, editor, *Proceedings of the Tenth ESSLLI Student Session*, pages 33–45, Edinburgh, Scotland, 2005.

[20] Mark Buckley and Christoph Benzmüller. A Dialogue Manager for the DIALOG Demonstrator. SEKI Report SR-04-06, Fachrichtung Informatik, Universität des Saarlandes, Saarbrücken, Germany, 2004.

[21] Mark Buckley and Christoph Benzmüller. System Description: A Dialogue Manager supporting Natural Language Tutorial Dialogue on Proofs. In David Aspinall and Christoph Lüth, editors, *Proceedings of the ETAPS Satellite Workshop on User Interfaces for Theorem Provers (UITP)*, pages 40–67, Edinburgh, Scotland, 2005.

[22] Alan Bundy. The Use of Explicit Plans to Guide Inductive Proofs. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction (CADE-9)*, volume 310 of *LNCS*, Argonne, Illinois, 1988. Springer.

[23] Lassaad Cheikhrouhou and Volker Sorge. $\mathcal{PDS}$— a three-dimensional data structure for proof plans. In *Proceedings of the International Conference on Artificial and Computational Intelligence for Decision, Control and Automation in Engineering and Industrial Applications (ACIDCA'2000)*, Monastir, Tunisia, 22–24 March 2000.

[24] Herbert H. Clark. *Using Language*. CUP, 1996.

[25] Herbert H. Clark and Emanuel F. Schaefer. Contributing to discourse. *Cognitive Science*, 13:259–294, 1989.

[26] Robin Cohen. A computational theory of the function of clue words in argument understanding. In *Proceedings of the 22nd annual meeting on Association for Computational Linguistics*, pages 251–258, NJ, USA, 1984. Association for Computational Linguistics.

[27] Robin Cooper, Stina Ericsson, Ian Lewin, and C.J Rupp. Dialogue Moves in Negotiative Dialogue. Technical Report Deliverable D1.2, Siridus, University of Gothenburg, 2000.

[28] Mark G. Core, Johanna D. Moore, and Claus Zinn. Supporting constructive learning with a feedback planner. In *Proceedings of the AAAI Fall Symposium: Building Dialogue Systems for Tutorial Applications*, Falmouth, MA, 2000. AAAI Press.

[29] Mark G. Core, Johanna D. Moore, and Claus Zinn. The role of initiative in tutorial dialogues. In *The 10th Conference of the European Chapter of the ACL(EACL)*, pages 67–74, Budapest, 2003.

[30] Ken Currie and Austin Tate. O-Plan: the open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

[31] George Ferguson, James Allen, and Brad Miller. Trains-95: Towards a mixed-initiative planning assistant. In *Proceedings of the Third Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pages 70–77, Edinburgh, 1996.

[32] Armin Fiedler. Dialog-driven adaptation of explanations of proofs. In Bernhard Nebel, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1295–1300, Seattle, WA, 2001. Morgan Kaufmann.

[33] Armin Fiedler and Malte Gabsdil. Supporting progressive refinement of Wizard-of-Oz experiments. In Carolyn Penstein Rosé and Vincent Aleven, editors, *Proceedings of the ITS 2002 — Workshop on Empirical Methods for Tutorial Dialogue Systems*, pages 62–69, San Sebastián, Spain, 2002.

[34] Armin Fiedler and Dimitra Tsovaltzi. Automating hinting in mathematical tutorial dialogue. In *Proceedings of the EACL-03 Workshop on Dialogue Systems: Interaction, Adaptation and Styles of Management*, pages 45–52, Budapest, 2003.

[35] Gerhard Fliedner and Daniel Bobbert. DiaMant: A Tool for Rapidly Developing Spoken Dialogue Systems. In *Proceedings of the 7th Workshop on the Semantics and Pragmatics of Dialogue (DiaBruck)*, Wallerfangen, Germany, 2003.

[36] Gerhard Fliedner and Daniel Bobbert. A framework for information-state based dialogue (demo abstract). In *Proceedings of the 7th workshop on the semantics and pragmatics of dialogue DiaBruck*, Saarbrücken, 2003.

[37] Gerhard Gentzen. Untersuchungen über das logische Schließen I & II. *Mathematische Zeitschrift*, 39:176–210, 572–595, 1935.

[38] Jonathan Ginzburg. Dynamics and the semantics of dialogue. In Jerry Seligman and Dag Westerstahl, editors, *Language, Logic and Computation, Volume 1*, CSLI Lecture Notes, pages 221–237. CSLI, Stanford, 1996.

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

[40] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Number 78 in Lecture Notes in Computer Science. Springer, 1979.

[41] Arthur C. Graesser, Katja Wiemer-Hastings, Peter Wiemer-Hastings, and Roger Kreuz. Autotutor: A simulation of a human tutor. *Cognitive Systems Research*, 1:35–51, 1999.

[42] Herbert P. Grice. Logic and conversation. *The Logic of Grammar*, pages 64–75, 1975.

[43] Barbara Grosz. *Understanding Spoken Language*, chapter Discourse Analysis, pages 235–268. Elsevier North-Holland, New York, 1978.

[44] Barbara J. Grosz, Aravind K. Joshi, and Scott Weinstein. Centering: A framework for modeling the local coherence of discourse. *Computational Linguistics*, 21(2):203–225, 1995.

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

[46] Michael A. K. Halliday and Ruqaiya Hasan. *Cohesion in English*. Longman, London, 1976.

[47] Peter A. Heeman and James Allen. The TRAINS 93 dialogues. Technical note 94-2, University of Rochester, Rochester, New York, 1995.

[48] Charles T. Hemphill, John Godfrey, and George R. Doddington. The ATIS spoken language systems pilot corpus. In *Proceedings DARPA Speech and Natural Language Workshop*, pages 96–101, Hidden Valley, PA, 1990. Morgan Kaufmann.

[49] Jerry R. Hobbs. Coherence and coreference. *Cognitive Science*, 3(1):67–90, 1979.

[50] Xiaorong Huang. Reconstructing proofs at the assertion level. In Alan Bundy, editor, *Proceedings of the 12th Conference on Automated Deduction*, pages 738–752. Springer, 1994.

[51] Malte Hübner, Serge Autexier, Christoph Benzmüller, and Andreas Meier. Interactive theorem proving with tasks. *Electronic Notes in Theoretical Computer Science*, 103(C):161–181, 2004.

[52] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Prentice Hall, New Jersey, USA, 2000.

[53] Hans Kamp and Uwe Reyle. *From Discourse to Logic*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1993.

[54] Michael Kohlhase and Andreas Franke. Mbase: Representing knowledge and context for the integration of mathematical software systems. *Journal of Symbolic Computation; Special Issue on the Integration of Computer Algebra and Deduction Systems*, 32(4):365–402, 2001.

[55] Staffan Larsson. *Issue-based Dialogue Management*. PhD thesis, Department of Linguistics, University of Gothenburg, Sweden, 2002.

[56] Oliver Lemon, Anne Bracy, Alexander Gruenstein, and Stanley Peters. Information States in a Multi-modal Dialogue System for Human-Robot Conversation. In *Proceedings of Bi-Dialog, 5th Workshop on Formal Semantics and Pragmatics of Dialogue*, pages 57 – 67, 2001.

[57] Stephen C. Levinson. *Pragmatics*. Cambridge University Press, Cambridge, 1983.

[58] David Lewis. Scorekeeping in a language game. *Journal of Philosophical Logic*, 8:339–359, 1979.

[59] Diane Litman, Michael Kearns, Satinder Singh, and Marilyn Walker. Automatic optimization of dialogue management. In *Proceedings of the 18th International Conference on Computational Linguistics (COLING)*, Saarbrücken, Germany, 2000.

[60] Diane J. Litman and Scott Silliman. ITSPOKE: An Intelligent Tutoring Spoken Dialogue System. In *Proceedings of the Human Language Technology Conference: 4th Meeting of the North American Chapter of the Association for Computational Linguistics (HLT/NAACL) (Companion Proceedings)*, Boston, MA, 2004.

[61] William C. Mann and Sandra A. Thompson. Rhetorical Structure Theory: A Theory of Text Organization. Technical Report RS-87-190, Information Sciences Institute, University of Southern California, 1987.

[62] David L. Martin, Adam J. Cheyer, and Douglas B. Moran. The Open Agent Architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1–2):91–128, 1999.

[63] Colin Matheson, Massimo Poesio, and David Traum. Modelling grounding and discourse obligations using update rules. In *Proceedings of the first conference of the North American chapter of the Association for Computational Linguistics*, Seattle, 2000.

[64] William McCune. Otter 3.0 Reference Manual and Guide. Technical Report ANL-94-6, Argonne National Laboratory, Argonne, Illinois 60439, USA, 1994.

[65] Michael F. McTear. Modelling spoken dialogues with state transition diagrams: Experiences with the CSLU toolkit. In *Proceedings of the 5th International Conference on Spoken Language Processing*, Sydney, Australia, 1998.

[66] Michael F. McTear. Spoken Dialogue Technology: Enabling the Conversational User Interface. *ACM Computing Surveys*, 34(1):90–169, 2002.

[67] Andreas Meier. *Proof Planning with Multiple Strategies*. PhD thesis, Saarland University, Saarbrücken, Germany, 2004.

[68] Erica Melis, Eric Andres, Andreas Franke, George Goguadse, Michael Kohlhase, Paul Libbrecht, Martin Pollet, and Carsten Ullrich. A generic and adaptive web-based learning environment. In *Artificial Intelligence and Education*, pages 385–407, 2001.

[69] Erica Melis and Jörg Siekmann. Knowledge-based proof planning. *Artificial Intelligence Journal*, 115(1):65–105, 1999.

[70] Johanna Moore. What makes human explanations effective? In *Proceedings of the Fifteenth Annual Meeting of the Cognitive Science Society*, pages 131–136, Hillsdale, NJ, 1993.

[71] NUANCE Developers' Toolkit. `http://www.nuance.com/`.

[72] Manfred Pinkal, Jörg Siekmann, and Christoph Benzmüller. Projektantrag Teilprojekt MI3 — DIALOG: Tutorieller Dialog mit einem mathematischen Assistenzsystem. In *Fortsetzungsantrag SFB 378 — Ressourcenadaptive kognitve Prozesse*, Universität des Saarlandes, Saarbrücken, Germany, 2001.

[73] Manfred Pinkal, Jörg Siekmann, Christoph Benzmüller, and Ivana Kruijff-Korbayova. Dialog: Natural language-based interaction with a mathematics assistance system. Project proposal in the Collaborative Research Centre SFB 378 on Resource-adaptive Cognitive Processes, 2004.

[74] Massimo Poesio and David Traum. Conversational actions and discourse situations. *Computational Intelligence*, 13(3), 1997.

[75] Massimo Poesio and David Traum. Towards an axiomatization of dialogue acts. In J. Hulstijn and A. Nijholt, editors, *Proceedings of TWENDIAL, the Twente Workshop on the Formal Semantics and Pragmatics of Dialogues*, pages 207–222, Enschede, 1998.

[76] Siridus project. `http://www.ling.gu.se/projekt/siridus/`.

[77] Trindi project. `http://www.ling.gu.se/research/projects/trindi/`.

[78] Aarne Ranta. Grammatical framework — a type-theoretical grammar formalism. *Journal of Functional Programming*, 14(2):145–189, 2004.

[79] Ehud Reiter and Robert Dale. Building applied natural language generation systems. *Journal of Natural Language Engineering*, 3(1):57–87, 1997.

[80] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence, New Jersey, 1995.

[81] Harvey Sacks, Emanuel A Schegloff, and Gail Jefferson. A simplest systematics for the organization of turn-taking for conversation. *Language*, 50(4):696–735, 1974.

[82] Emanuel A. Schegloff. Sequencing in conversational openings. *American Anthropologist*, 70:1075–1095, 1968.

[83] John R. Searle. *Speech Acts*. Cambridge University Press, New York, 1969.

[84] John R. Searle. A taxonomy of illocutionary acts. *Language, Mind and Knowledge, Minnesota Studies in the Philosophy of Science*, pages 344–369, 1975.

[85] Jörg Siekmann and Christoph Benzmüller. Omega: Computer supported mathematics. In *Proceedings of the 27th German Conference on Artificial Intelligence (KI 2004)*, Ulm, Germany, 2004.

[86] Jörg Siekmann, Christoph Benzmüller, Armin Fiedler, Andreas Meier, Immanuel Normann, and Martin Pollet. Proof development in OMEGA: The irrationality of square root of 2. In Fairouz Kamareddine, editor, *Thirty Five Years of Automating Mathematics*, Kluwer Applied Logic series (28), pages 271–314. Kluwer Academic Publishers, 2003.

[87] Ronnie W. Smith and D. Richard Hipp. *Spoken natural language dialog systems: a practical approach*. Oxford University Press, Inc., New York, NY, USA, 1995.

[88] Volker Sorge. *Ω-Ants — A Blackboard Architecture for the Integration of Reasoning Techniques into Proof Planning*. PhD thesis, Department of Computer Science, Saarland University, Saarbrücken, Germany, 2001.

[89] CLT Sprachtechnologie. `http://www.clt-st.de/`.

[90] Robert C. Stalnaker. Assertion. *Syntax and Semantics*, 9:315–332, 1978.

[91] TALK Project. `http://www.talk-project.org/`.

[92] CSLU Toolkit. `http://cslu.cse.ogi.edu/toolkit/`.

[93] David Traum. *A computational theory of grounding in natural language conversation*. PhD thesis, University of Rochester, NY, USA, 1994. Also available as TR 545, Dept. of Computer Science, University of Rochester.

[94] David Traum, Johan Bos, Robin Cooper, Staffan Larsson, Ian Lewin, Colin Matheson, and Massimo Poesio. A model of dialogue moves and information state revision. Technical report TRINDI project deliverable D2.1, University of Gothenburg, 1999.

[95] David Traum and Staffan Larsson. The information state approach to dialogue management. In J. van Kuppevelt and R. Smith, editors, *Current and new directions in discourse and dialogue*. Kluwer, 2003.

[96] Dimitra Tsovaltzi and Elena Karagjosova. A view on dialogue move taxonomies for tutorial dialogues. In *Proceedings of 5th SIGdial Workshop on Discourse and Dialogue*, Boston, USA, 2004.

[97] Kurt VanLehn, Pamela W. Jordan, Carolyn P. Rosé, Dumisizwe Bhembe, Michael Boettner, Andy Gaydos, Maxim Makatchev, Umarani Pappuswamy, Michael Ringenberg, Antonio Roque, Stephanie Siler, and Ramesh Srivastava. The Architecture of Why2-Atlas: A Coach for Qualitative Physics Essay Writing. In *Proc. 6th Int. Conf. on Intelligent Tutoring Systems*, volume 2363 of *LNCS*, pages 158–167. Springer, 2002.

[98] VoiceXML Forum. `http://www.voicexml.org/`.

[99] Wolfgang Wahlster, editor. *Verbmobil: Foundations of Speech-to-Speech Translation.* Springer, Berlin, Germany, 2000.

[100] Magdalena Wolska, Bao Quoc Vo, Dimitra Tsovaltzi, Ivana Kruijff-Korbayova, Elena Karagjosova, Helmut Horacek, Malte Gabsdil, Armin Fiedler, and Christoph Benzmüller. An annotated corpus of tutorial dialogs on mathematical theorem proving. In *Proceedings of International Conference on Language Resources and Evaluation (LREC 2004)*, Lisbon, Potugal, 2004. ELDA.

[101] Claus Zinn, Johanna D. Moore, and Mark G. Core. A 3-tier planning architecture for managing tutorial dialogue. In *Proceedings of Intelligent Tutoring Systems, Sixth International Conference*, Biarritz, France, 2002.

[102] Claus Zinn, Johanna D. Moore, Mark G. Core, Sebastian Varges, and Kaśka Porayska-Pomsta. The BE&E Tutorial Learning Environment (BEETLE). In *Proceedings of Diabruck, the 7th Workshop on the Semantics and Pragmatics of Dialogue*, Saarbrücken, Germany, 2003.