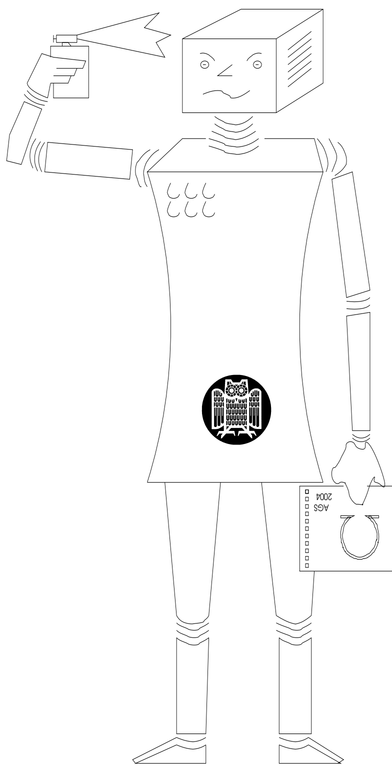
SEKI ISSN 1437-4447



A Dialogue Manager for the DIALOG Demonstrator

Mark Buckley and Christoph Benzmüller
Dept. of Computer Science, Saarland University
{markb|chris}@ags.uni-sb.de

SEKI Report SR–2004–06

A Dialogue Manager for the DIALOG Demonstrator

Mark Buckley and Christoph Benzmüller
Dept. of Computer Science, Saarland University
{markb|chris}@ags.uni-sb.de

January 22, 2005

Abstract

The DIALOG project investigates flexible natural language tutorial dialogue in mathematics, and as such it is essential that the system includes a dedicated module to facilitate and control interaction with the tutor. In this report we present the design and implementation of the dialogue manager for the demonstrator system of the DIALOG project. We begin with a brief survey of some approaches to dialogue management, followed by an overview of the system and a description of the modules which are part of the demonstrator. The dialogue manager is an information state update based manager built on a platform for dialogue management applications called Rubin. We describe the functionality of the Rubin platform before giving the specification of the dialogue manager itself, including its input rules and the structure of the information state. Finally we discuss some aspects of the system, its implementation, and the pros and cons of using the Rubin platform.

1 Introduction

In this report we present and discuss the design and implementation of the dialogue manager for the demonstrator system of the DIALOG project[1] [4, 16, 17, 18]. This system was built for demonstration purposes by the DIALOG team[2] at Saarland University for the review of the Collaborative Research Centre 378 on June 8, 2004. The goal of the DIALOG project is to investigate flexible natural language tutorial dialogue in mathematics; our particular focus is on tutoring mathematical proofs in naive set theory. Since the medium of communication is natural language dialogue, and since tutorial dialogues are by nature both flexible and unpredictable (from the standpoint of the tutor), it is essential to include a sophisticated, dedicated dialogue manager to handle the interaction between student and the system modules.

There are a number of candidate approaches to dialogue management which could be suitable for DIALOG. Finite-state methods, such as the CSLU toolkit [13, 24], are suited to situations

[1] `http://www.ags.uni-sb.de/~chris/dialog/`, `http://www.coli.uni-sb.de/sfb/`

[2] The DIALOG team is: Christoph Benzmüller, Ivana Kruijff-Korbayova, Manfred Pinkal, Jörg Siekmann, Dimitra Tsovaltzi, Quoc Bao Vo, Magdalena A. M. Wolska, Serge Autexier, Armin Fiedler, Erica Melis, Beata Biehl, Mark Buckley, Oliver Culo, Sreedhar Ellisetty, Hussain Syed Sajjad, Andrea Schuh, Jochen Setz, Michael Wirth.

where a certain set of data must be collected by an agent in order to carry out some action, or where the number of possible dialogues is relatively small. Such systems are characterised by a finite state machine which contains all possible dialogues; dialogues are hard–wired and system–driven. Such methods are not sufficient for DIALOG because of our interest in flexible, natural tutorial dialogues.

The form-filling approach, such as in the AUTOTUTOR system [9], is more adaptable than finite-state. The information that the system seeks is stored in slots in a form which is incrementally filled until the required amount of information is reached. This allows the system to be more flexible in relation to the order in which information is elicited from the user. However, even this flexibility does not reach the level required by DIALOG. Also, form-filling is more suited to situations in which the information flow is mainly in the direction of the system, for instance in personal banking applications, whereas the dialogue manager for DIALOG must support flexible information exchange in both directions.

The solution we decided on is the Information State Update approach. In this design the dialogue manager maintains a description of the state of the discourse and its participants, which then forms a framework for communication between the external modules associated with the system.

This report begins with an outline of the functionality that a dialogue manager for the DIALOG project should support and a brief description of the overall architecture. We then describe Rubin, the development platform that the dialogue manager is built on, followed by the specification of the dialogue manager itself. Finally we discuss some aspects of the manager, of Rubin, and of system development as a whole.

1.1 The Sample Dialogue

The DIALOG demonstrator has been developed to illustrate the functionality of the DIALOG system at hand of a few dialogues from the project's Wizard-of-Oz corpus [6]. Here we concentrate on dialog did16k. The task that the student is asked to prove is theorem (1).

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

The examples in this report of information exchange between modules are all taken from this sample dialogue.

Some of the modules are still only simulated in our demonstrator; our emphasis so far has been on the development of the input analyser, proof manger, tutorial manager, natural language generator and graphical user interface.

2 The Dialogue Manager

The function of the dialogue manager in DIALOG is to handle interaction between student and system, and to facilitate communication between system modules. The following modules are connected to the system (see also Figure 1):

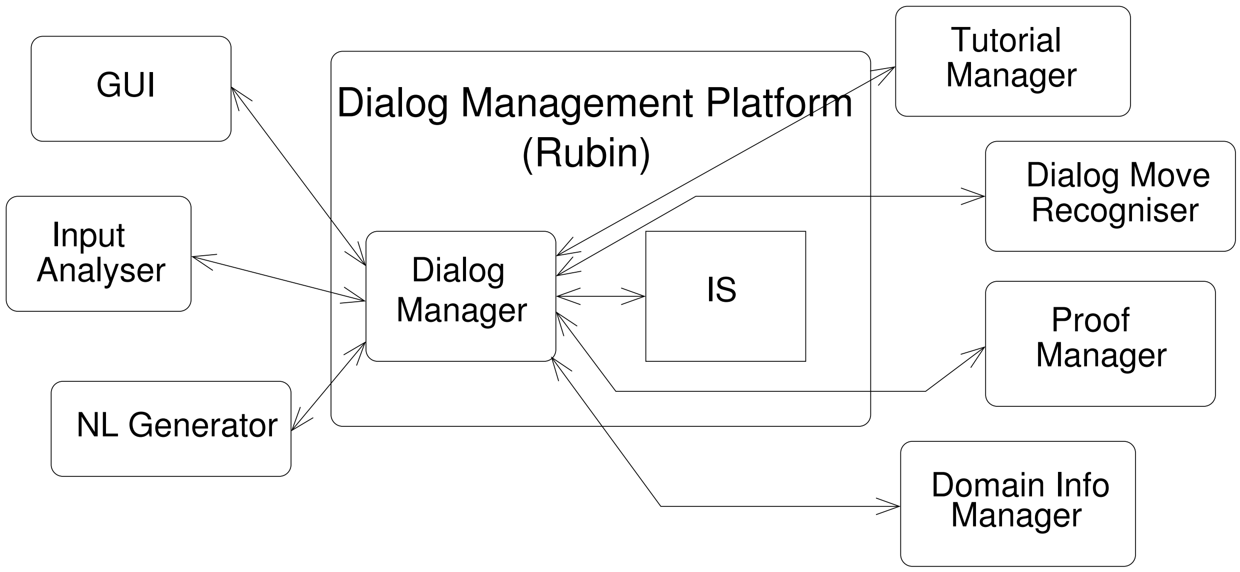


Figure 1: Architecture

GUI The graphical user interface that the student uses. So far we assume only typed input in the project.

Input Analyser This is the sentence analyser which parses the student's utterance and determines its linguistic content and an underspecified representation of the proof step (the mathematical content of the utterance) that the student performed, including for instance the formula in the utterance and the type of inference.

Dialog Move Recogniser This module identifies the function (i.e. dialogue moves) of utterances, based on the current state of the dialogue.

Proof Manager The proof manager is based on ΩMEGA–CORE, and evaluates student input and monitors and maintains the proof state.

Domain Info Manager This module uses mathematical and tutorial knowledge to determine the mathematical information of the proof step at hand, which is potentially relevant to the tutorial manager and the natural language generator.

Tutorial Manager The tutorial manager provides and applies pedagogical knowledge which specifies generic and domain–specific teaching strategies, including didactic and socratic teaching methods, and hinting dialog moves.

NL Generator This module creates a natural language realisation of the dialogue move with which the system responds.

The system modules will be described in detail in Section 3.

The design of the dialogue manager is based on the Information State Update approach used in the SIRIDUS and TRINDI projects [19, 20, 26], and implemented in TrindiKit [25]. The Information State (IS) is a central data structure storing information about the current state of the dialog and about the internal states of modules participating in the dialogue. It is divided into "private" system information, "public" information shared between the system and the user, and information which is neither private nor fully public, but rather shared between certain system modules. The IS stores both dialogue-level knowledge, such as the user's last speech act or an evaluation of the utterance, as well as meta–information about the dialogue, such as an utterance history.

2.1 Dialogue Move Selection

At its simplest level, the function of a dialogue manager is to receive a dialogue move from the linguistic analysis module, and, based on the current IS, decide on the most appropriate dialogue move to respond with. A dialogue move is a notion which extends that of a speech act. It consists of a number of dimensions, each of which encode a different aspect of the information contained in the utterance. For example, the forward-looking dimension corresponds to the notion of a speech act, and the backward-looking dimension accounts for the relationship of the utterance to the dialogue up to that point. Dimensions can themselves contain hierarchical structure. In this way a single dialogue move can account for the many functions that an utterance may have. Consider example (2) from the corpus of the DIALOG Wizard-of-Oz experiments [28] (translated from German):

(2) "Can you explain that in more detail?"

This utterance is a request for information, it refers back to a previous utterance (the anaphor "that") which the system made, and it introduces an obligation on the system to explain that utterance.

Each of the functions of an utterance are encoded in the dimensions of its dialogue move. The taxonomy of dialogue moves used in DIALOG is described in [27]; it is an adaptation and extension of the DAMSL taxonomy [1]. DAMSL is a standard and application–independent annotation scheme for dialogue tagging, and the taxonomy used in DIALOG is therefore 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 utterances that address the management of the task at hand, for instance beginning a case distinction or declaring a proof complete.

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

What dialogue move the system produces is determined based on information supplied by each of the modules mentioned above. 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 its proof step, 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, if it is relevant, etc. 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 the user with 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 response dialogue move, along with any extra specifications required, to the generation module to be verbalised, the resulting utterance is outputted, and the turn passes to the user. At this point the system waits for the next user utterance to be received. This results in a sequence of dialogue moves according to the model of the dialogue.

2.2 Inter–module communication

The second responsibility of the dialogue manager is to be 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. Because the dialogue manager is based on information state updates, each result computed by a module needs to be stored in the IS. 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 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. For instance, to make appropriate decisions on hinting moves in the case of an uttered proof step, the tutorial manager needs to know the evaluation of the proof step; i.e. whether it was correct, incorrect, correct but irrelevant, etc. This information is passed through the dialogue manager and the IS. First it is stored in the IS having been received from the proof manager. When the tutorial manager needs this piece of information, it can be read by the dialogue manager from the IS and sent to the tutorial manager.

Here we would like to stress that there are 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. This is the view that was introduced in Section 2.1. The other notion of a dialogue manager is a platform which supports the development of a dialogue-based application, often one which implements the information-state approach. 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 section 2.2. The DIALOG demonstrator contains subsystems which fulfil both of these tasks, and in this report we concentrate on the second notion of dialogue management – the development platform for dialogue applications.

2.3 Implementation Platform

The dialogue manager is built on Rubin [8], a platform for developing dialogue management applications from CLT company, which is described in section 4.

3 Modules connected to the DIALOG System

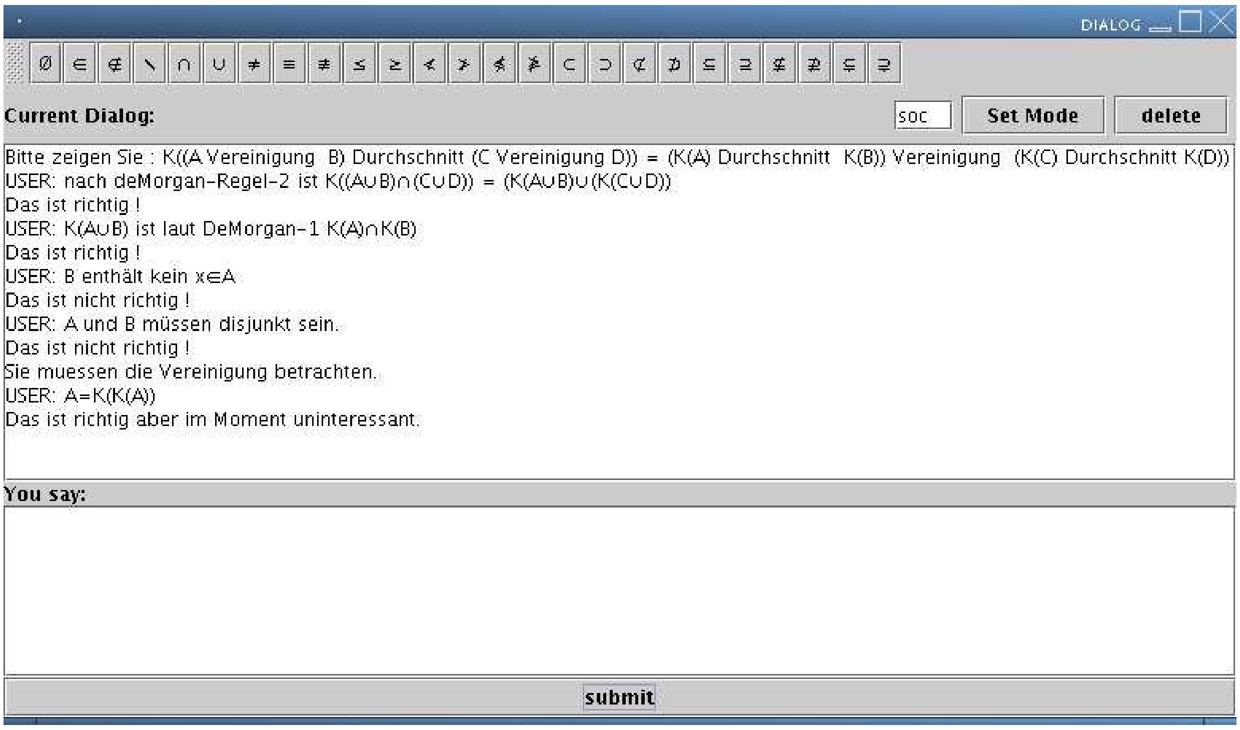
As shown in the diagram of the system in Figure 1, the dialogue manager acts as the communications centre for each module that is connected to it, and it in turn accesses the information state. In this section we detail the functions of each of the seven modules which are connected to the dialogue manager. Information enclosed in chain brackets represents a structure, information in round brackets is a list. See section 4.1 for details of the Rubin data structures. 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 ∪ B) ∩ (C ∪ D)) = (K (A ∪ B) ∪ K (C ∪ D))”; see also Figure 2.

3.1 Graphical User Interface

The GUI of the demonstrator program is an extension of the DiaWoZ tool [6], which has been developed in the DIALOG project at the very beginning to support the Wizard-of-Oz experiments in which we collected our corpus. The GUI is presented in Figure 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. In the GUI for the demonstrator there are two extra buttons and an input field, as shown in the Figure. These are used to set the tutorial mode, i.e. minimal, 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:[3] A string (the system utterance), which is then displayed in the conversation field, e.g.:

[3]The notion of input/output depends on point of view: the results that a module computes are its output, which



```
Bitte zeigen Sie : K((A Vereinigung  B) Durchschnitt (C Vereinigung D)) = (K(A) Durchschnitt  K(B)) Vereinigung  (K(C) Durchschnitt K(D))
USER: nach deMorgan-Regel-2 ist K((A∪B)∩(C∪D)) = (K(A∪B)∪(K(C∪D))
Das ist richtig !
USER: K(A∪B) ist laut DeMorgan-1 K(A)∩K(B)
Das ist richtig !
USER: B enthält kein x∈A
Das ist nicht richtig !
USER: A und B müssen disjunkt sein.
Das ist nicht richtig !
Sie muessen die Vereinigung betrachten.
USER: A=K(K(A))
Das ist richtig aber im Moment uninteressant.
```

Figure 2: The DiaWoz tool, extended for the DIALOG demonstrator, showing the first 5 moves of the sample dialogue.

"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:

```
{  st_input  =  "Nach deMorgan-Regel-2 ist K ( ( A ∪ B ) ∩ ( C ∪ D ) ) = ( K ( A ∪ B ) ∪
                K ( C ∪ D ) )"
   mode      =  "min",
   delete    =  false }
```

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 [15], and its linguistic meaning is represented using *Hybrid Logic Dependence Semantics* (HDLS) [3].

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

Output: A structure containing the linguistic meaning (LM) represented in HDLS and the underspecified proof step contained in the utterance, in an ad-hoc LISP-like representation (LU). This is a language in the spirit of the proof representation language described in [2], but designed for the inter-module communication requirements of the DIALOG project:

then become the input to the dialogue manager. In this section we take the point of view of the module, that is, 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.

{ LM = $@h1$(holds $\wedge$ <CRITERION>($d1$ $\wedge$ deMorgan-Regel-2) $\wedge$ <PATIENT>($f1$ $\wedge$ FORMULA))

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))))
}
```

3.3 Dialog 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 outputted 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 utterance:

{ 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 session. 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.

3.4 Proof Manager

The proof manager is the mediator between the dialogue manager and the mathematical proof assistant ΩMEGA–CORE [21, 22]. 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 [12] and the user model in ActiveMath [14], 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 [2, 11].

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 [2] (LU in the output of the input analyser (3.2)). 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 outputted by the input analyser (LU in Section 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 "Proof-StepCompleted false", meaning that after this proof step has been integrated into the student's partial proof plan, the proof is still not complete.

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"}
```

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 [7], 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, or "soc" for socractic.

Input: The tutorial mode, the task dimension of the user's dialogue move, which is determined by the dialog 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.

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 [27]) 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.

3.7 NL Generator

The natural language generation system used in DIALOG is *P.rex* [5]. *P.rex* is designed to present complete proofs in natural language, and has been 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!”.

4 Rubin

Rubin is a platform for building dialogue management applications developed by the CLT company [23]. 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.

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 consists of the following sections:

dialogue_model	:=	*IS*
		*device_declaration**
		[*grammar*]
		*support_function**
		*plan**
		*input_rule**

Information State The IS 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 datatypes: `bool`, `int`, `real`, `string`, `list` or `struct`. The IS is specified by the following syntax:

IS	:=	*slot**
slot	:=	*label* [: *type*][= *value*]
type	∈	{`bool`, `int`, `real`, `string`, `list`, `struct`}

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:

list	:=	[ ] | [*value* {, *value*}*]
struct	:=	{*slot**}

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 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 }

In DIALOG we do not make use of a grammar, since input analysis is handled by our own input analyser.

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 datatypes used in the dialogue model, and the syntax is nearly identical to ANSI C:

support_function := {*type* | *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, 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:

plan	:=	*name*({*type label*}*)
		preconditions
		{*statement** }
preconditions	:=	[ ] | [*precondition* {, *precondition*}*]
precondition	:=	*pos_precon* | *neg_precon*
pos_precon	:=	: *condition*
neg_precon	:=	!: *condition*
condition	:=	*slot_name* { == | != } *value*

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

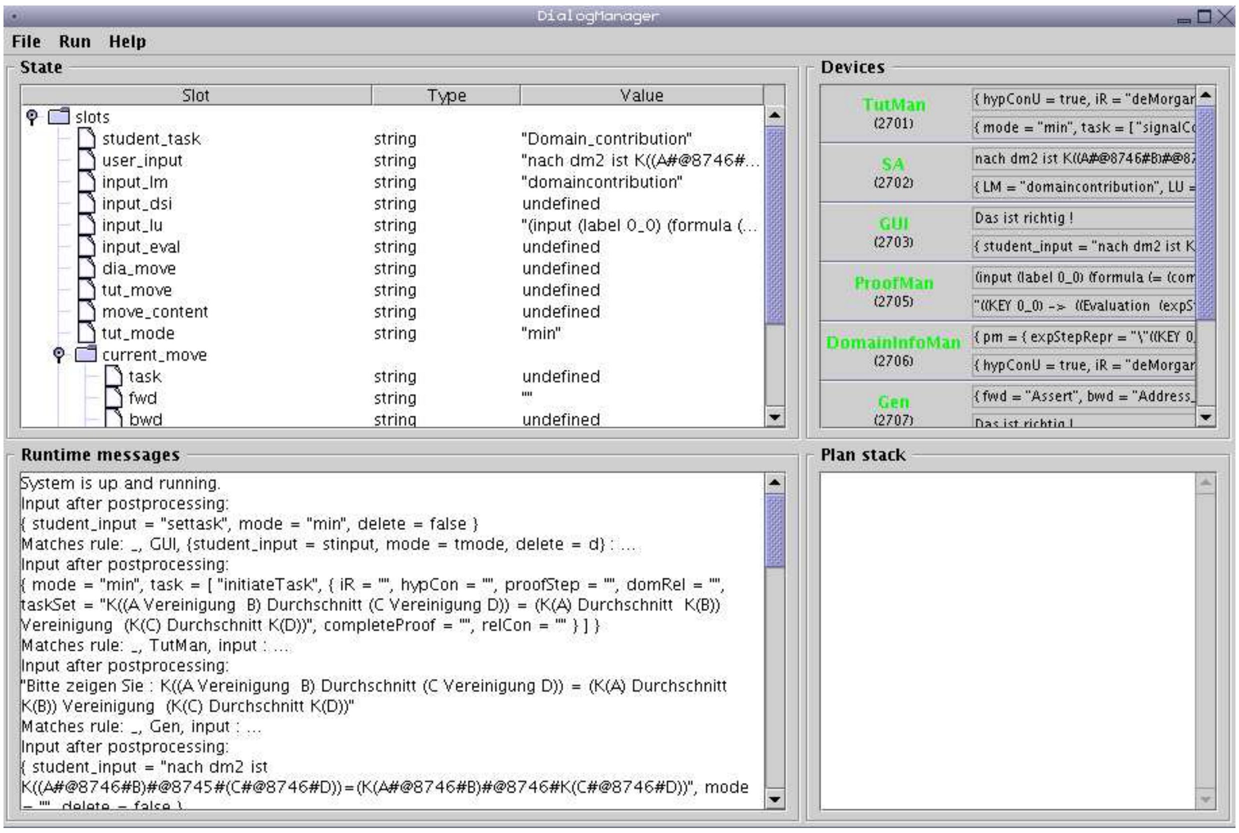
```
{ x = 3 }
```

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[4] 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.

[4]Here we speak of matching as opposed to unification. In Rubin it is not possible to have variables in the information state, so matching is sufficient to decide on the applicability of rules and to bind input to local variable names.



Slot	Type	Value
slots		
student_task	string	"Domain_contribution"
user_input	string	"nach dm2 ist K((A#@8746#...
input_lm	string	"domaincontribution"
input_dsi	string	undefined
input_lu	string	"(input (label 0_0) (formula (...
input_eval	string	undefined
dia_move	string	undefined
tut_move	string	undefined
move_content	string	undefined
tut_mode	string	"min"
current_move		
task	string	undefined
fwd	string	""
bwd	string	undefined

TutMan (2701)	{ hypConU = true, iR = "deMorgar
	{ mode = "min", task = ["signalCo
SA (2702)	nach dm2 ist K((A#@8746#B)#@87
	{ LM = "domaincontribution", LU =
GUI (2703)	Das ist richtig !
	{ student_input = "nach dm2 ist K
ProofMan (2705)	(input (label 0_0) (formula (= (com
	"((KEY 0_0) -> ((Evaluation (expS
DomainInfoMan (2706)	{ pm = { expStepRepr = "\"((KEY 0_
	{ hypConU = true, iR = "deMorgar
Gen (2707)	{ fwd = "Assert", bwd = "Address_
	Das ist richtig !

```
System is up and running.
Input after postprocessing:
{ student_input = "settask", mode = "min", delete = false }
Matches rule: _, GUI, {student_input = stinput, mode = tmode, delete = d} : ...
Input after postprocessing:
{ mode = "min", task = [ "initiateTask", { iR = "", hypCon = "", proofStep = "", domRel = "",
taskSet = "K((A Vereinigung  B) Durchschnitt (C Vereinigung D)) = (K(A) Durchschnitt  K(B))
Vereinigung  (K(C) Durchschnitt K(D))", completeProof = "", relCon = "" } ] }
Matches rule: _, TutMan, input : ...
Input after postprocessing:
"Bitte zeigen Sie : K((A Vereinigung  B) Durchschnitt (C Vereinigung D)) = (K(A) Durchschnitt
K(B)) Vereinigung  (K(C) Durchschnitt K(D))"
Matches rule: _, Gen, input : ...
Input after postprocessing:
{ student_input = "nach dm2 ist
K((A#@8746#B)#@8745#(C#@8746#D))=(K(A#@8746#B)#@8746#K(C#@8746#D))", mode
= "", delete = false }
```

Figure 3: The Rubin GUI, showing the beginning of a session with the DIALOG demonstrator.

The most precise formal definition of an input rule that is available, taken from a presentation made by the designers of the Rubin system, is that "rule bodies are just inlined plans that can update 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:

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

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 IS slots which satisfies the IS constraints of the rule and $input$ matches with the input pattern of the rule.

4.2 Rubin's Graphical User Interface

The Rubin graphical user interface shows details of all communication to and from the Rubin server and the current values in the IS (see Figure 3). The current values of the IS slots are

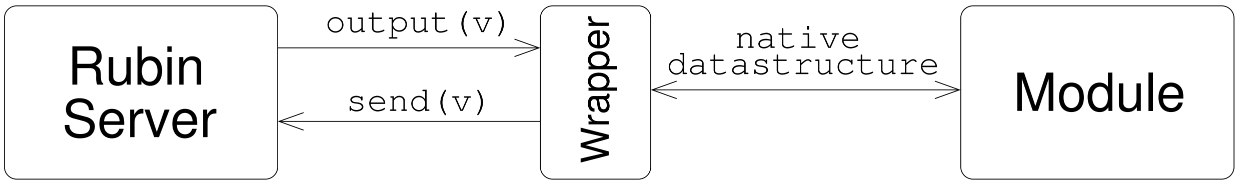


Figure 4: Wrapper communication between Rubin and a module.

displayed in the upper left window, and in this windows it is possible to alter the values of slots in-place at runtime. This is useful for rapid prototyping, 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 is 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.

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)` (see Figure 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 is written in ordinary 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 operation 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 The Dialogue Model

The dialogue model for the DIALOG demonstrator contains the specification of the IS slots, the device connections, the update rules to capture input from modules, and plans for unparsable input. The slots that make up the IS in our example are listed in Table 1.

The modules described in Section 3 are assigned a port number for socket communication:

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

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 sentence analyser "domaincontribution"
input_lu	`string`	Underspecified representation of the user's proof step (input (label 1_1) (formula ...) (see Section 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 3.3)
complete_proof	`string`	The complete user proof, outputted by the proof manager when the proof has been completed. ((KEY 1_1) → ((Evaluation (expStepRepr (label 1_1) (formula ...) (see Section 3.4)
deleting	`bool`	A flag which is set to true when the latest user/system turn is to be undone. (true/false)

Table 1: The IS slots in the dialogue model.

```
DMR               : 2710;
```

Since our linguistic analysis is performed entirely by a more advanced sentence 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 still 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. This move is underspecified in the sense that the pedagogical knowledge of the tutorial manager has not yet been added.

Plans are used to handle unparsable input, since in this case no mathematical or pedagogical knowledge is required by the dialogue manager, and these modules therefore do not need to be called. When the dialogue manager receives input from the sentence 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.

```
_, "GUI", { student_input = stinput, mode = tmode, delete = d} : {
    slots.user_input = stinput;
    //check if the tutorial task is being set
    if (stinput == "settask") {
       slots.tut_mode = tmode;
       // send complete structure with null values to TutMan
       output_struct(TutMan, GUI, {delete = false, mode = tmode, ...});
    }else if (d==true) {
       //what to do if delete
       slots.deleting = true;
       output_string(SA, GUI, stinput);
    }else {
       if (tmode != "")
          slots.tut_mode = tmode;
       output_string(SA, GUI, stinput);
    }
}
```

Figure 5: The input rule for data received from the GUI.

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 IS. 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.

As an example consider the rule for input from the GUI, shown in Figure 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.

5.2 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 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[5]) 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 discourse level knowledge, which is encoded in the dialogue manager, the input analyser, dialogue move recogniser, and the NL generator.

What is not so clear from the information flow diagram is that each arrow 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 instance, when the dialogue move recogniser outputs data, the dialogue manager performs the "is domain contribution?" test, and based on this, passes control to either the proof manager or to itself.

[5]Proof step manager, a previous name for the domain information manager.

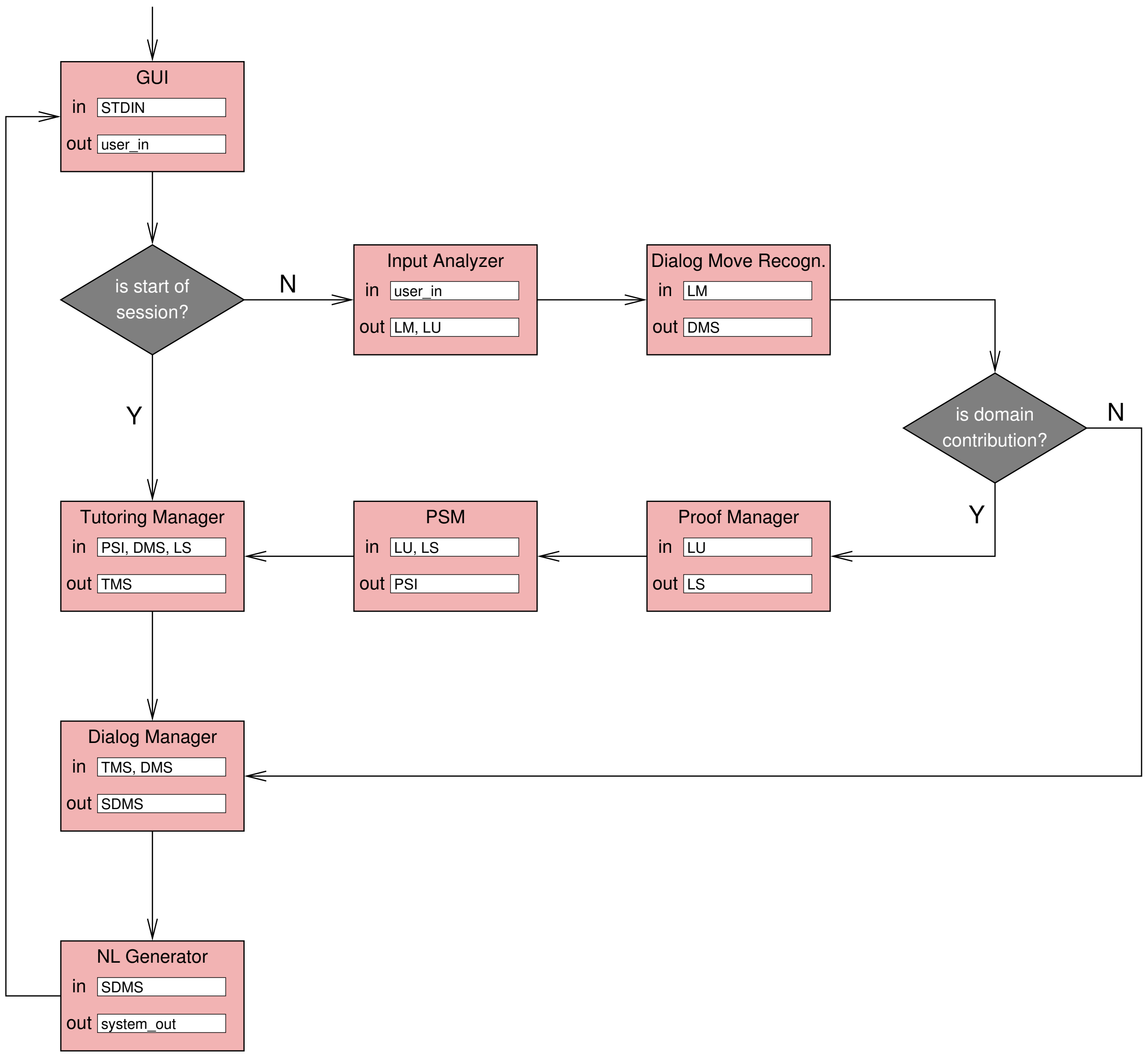


Figure 6: Information flow in the DIALOG demonstrator for a single system turn.

6 Discussion

6.1 Module Simulation

Because the DIALOG demonstrator is at a relatively early stage of development, a number of modules which are 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 sentence 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 task dimension of the dialogue move.

The dialogue move recogniser receives the linguistic meaning from the sentence 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 made module simulation quite straightforward, 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 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.

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 sentence analyser is written in both Java and Perl, and the tutorial manager and proof manager, as well as the MPA 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 are also 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 must be stopped, and can only be restarted when the operating system has 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 sentence analyser was the use of OpenCCG in Linux. Development of the sentence 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 sentence analyser. As a consequence of this the sentence analyser was run separately to the rest of the system for demonstration purposes.

6.3 Using Rubin

Using the Rubin tool as the platform to build the dialogue manager on had a number of advantages and disadvantages.

Advantages

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.

Disadvantages

The Rubin tool turns out not to support certain functionalities which are necessary for application of ISU-based dialogue management to the domain of tutorial dialogue. In a dialogue manager built on Rubin, it is not the IS which is the driving force of system behaviour, but rather the rules that accept module input. This is because modules have no direct read or write access to IS slots, and the only way to invoke system action is by causing a rule to fire. This has a number of consequences.

Flow of control lies in input rules
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 execution does not stop. This is necessary because the intelligence of the DIALOG system lies in the modules, not in the dialogue model, and it is the modules which carry out the real computation. The set of input rules thus forms a chain of invocations, and the sequence of actions within the dialogue manager for each turn is quite rigid. At all times only one module is executing, and 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.

An example of the inflexibility of the input rules is shown in Figure 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

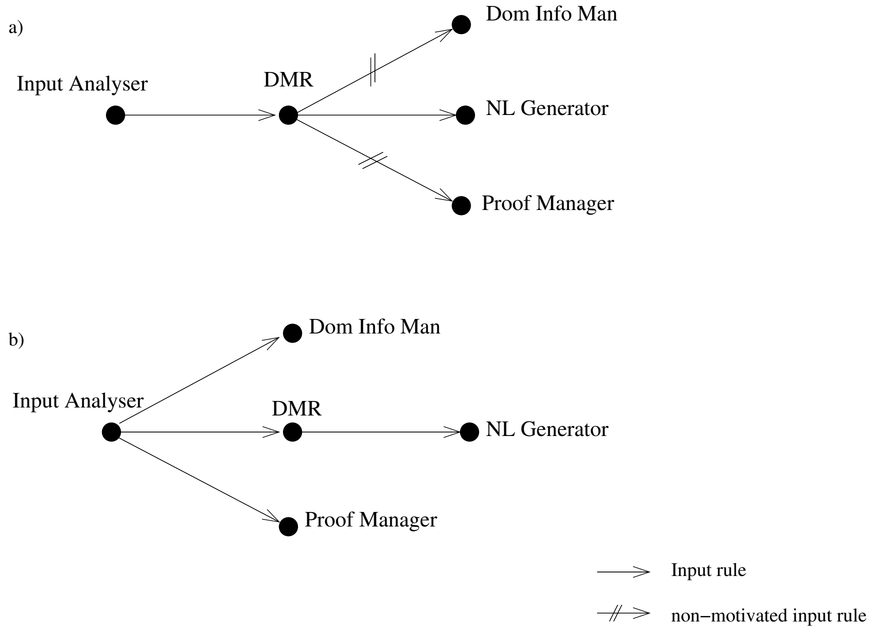


Figure 7: A section of the information flow, showing (a) scientifically less well motivated input rules and (b) the equivalent well-motivated input rules.

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.

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.

No IS update triggers

In Rubin there are no triggers on IS updates. That is, there is no way to state, "When slot A in the IS is updated (by any rule, plan, function), read its new value and take some conditional action", or even "When slot A is written, 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.

No runtime changes to the dialogue model
One other cause of inflexibility in Rubin is the static nature of the dialogue model. The IS, 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 change how the dialogue manager responds to certain input while the system is executing. This would be a desirable feature for example to rearrange the ordering of input rules, or to alter the IS constraints in the header of a rule. It is also not possible to register new modules or exchange modules at runtime. This could be a very useful feature for a dialogue system, as it would then be possible for instance to change language from German to English by replacing the natural language generation module, or to dynamically add and remove mathematical databases depending on the domain which is being taught.

Statically defined plans
Part of the dialogue model is a definition of dialogue plans. Plans are statically defined along with their preconditions before the dialogue is run, and cannot be changed at runtime. While this is a good approach to information-gathering or database-linked dialogues, it is not very 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 [29]. Due to the static nature of the plans it is not possible to refine dialogue planning at runtime in Rubin, and for tutorial dialogues, it is not possible to drive the system by dialogue plans alone.

No meta-level to control rule firing
In Rubin there is no meta-level which controls execution of input rules. The choice of input rule is determined by the order in which the rules are declared in the dialogue model, and as described above, this order is predefined and cannot be changed at runtime. To achieve a more flexible and adaptable control over rule firing, a meta-level is necessary in which more sophisticated criteria could be used to choose the most appropriate next rule. This is not possible in the current demonstrator. 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 the design of input rules, because there would be no need to decide on the "next step" of the system within the rule itself.

In summary, it turns out that a dialogue manager built on Rubin is not truly ISU based, because its behaviour is not determined by events in the IS. Rather the system behaviour is determined by its input rules. Coupled with the fact that modules have no access to the IS slots, the IS becomes little more than a storage area for the dialogue manager, rather than providing the impetus for system action, and the dialogue manager is left to rely on sequential calls to modules in order to collect all the necessary information to create a dialogue move.

6.4 Summary

In this report we have presented the design and implementation of a dialogue manager based on Rubin for the DIALOG demonstrator program. As a motivation for the implemented version we described the basic functionality the dialogue manager should provide, and described its role in the overall system. We introduced the Rubin tool and its characteristics, and presented the implementation of the dialogue manager within this framework. In the final section we discussed some aspects of the concrete system, as well as some pros and cons of using the Rubin platform.

Acknowledgements

This work was supported in part by the DAAD (German Academic Exchange Service), grant number A/03/15283, and by the Collaborative Research Centre 378 for Resource-adaptive Cognitive Processes.

We would like to thank all of the members of the DIALOG team for their input into this report, and for their helpful suggestions and comments on initial drafts.

References

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

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

[3] 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.

[4] Christoph Benzmüller, Armin Fiedler, Malte Gabsdil, Helmut Horacek, Ivana Kruijff-Korbayova, Manfred Pinkal, Jörg Siekmann, Dimitra Tsovaltzi, Bao Quoc Vo, and Magdalena Wolska. Tutorial dialogs on mathematical proofs. In *Proceedings of IJCAI-03 Workshop on Knowledge Representation and Automated Reasoning for E-Learning Systems*, pages 12–22, Acapulco, Mexico, 2003.

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

[6] 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.

[7] 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.

[8] 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.

[9] A. C. Graesser, K. Wiemer-Hastings, P. Wiemer-Hastings, and R. Kreuz. Autotutor: A simulation of a human tutor. *Cognitive Systems Research*, 1:35–51, 1999.

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

[11] Malte Hübner, Serge Autexier, Christoph Benzmüller, and Andreas Meier. Interactive theorem proving with tasks. *Electronic Notes in Theoretical Computer Science*, 103, 2004. To appear.

[12] M. Kohlhase and A. 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, September 2001.

[13] Michael 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.

[14] E. Melis, E. Andres, A. Franke, G. Goguadse, M. Kohlhase, P. Libbrecht, M. Pollet, and C. Ullrich. A generic and adaptive web-based learning environment. In *Artificial Intelligence and Education*, pages 385–407, 2001.

[15] openCCG. `http://openccg.sourceforge.net/`.

[16] 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.

[17] Manfred Pinkal, Jörg Siekmann, and Christoph Benzmüller. Dialog: Tutorial dialog with an assistance system for mathematics. Project report in the Collaborative Research Centre SFB 378 on Resource-adaptive Cognitive Processes, 2004.

[18] 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.

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

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

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

[22] Jörg Siekmann, Christoph Benzmüller, Armin Fiedler, Andreas Meier, Immanuel Normann, and Martin Pollet. *Proof Development in OMEGA: The Irrationality of Square Root of 2*, pages 271–314. Kluwer Applied Logic series (28). Kluwer Academic Publishers, 2003. ISBN 1-4020-1656-5.

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

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

[25] 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.

[26] 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. `http://www.ict.usc.edu/~traum/Papers/traumlarsson.pdf`.

[27] 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.

[28] M. Wolska, B. Quoc Vo, D. Tsovaltzi, I. Kruijff-Korbayova, E. Karagjosova, H. Horacek, M. Gabsdil, A. Fiedler, and C. 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.

[29] 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.