A Dialogue Manager Supporting Natural Language Tutorial Dialogue on Proofs

Mark Buckley[1] Christoph Benzmüller[2]

Dept. of Computer Science
Saarland University

Abstract

The DIALOG project investigates flexible natural language tutorial dialogue on mathematical proofs. Due to the flexible and unpredictable nature of tutorial dialogue in natural language it is essential to include a sophisticated, dedicated dialogue manager to handle the interaction between student and the system modules. In this paper we present the design and implementation of the dialogue manager for the demonstrator system of the DIALOG project. The dialogue manager forms the interface between the user and the system modules, including the automated theorem prover ΩMEGA–CORE, the tutorial module and the linguistic analysis module. We also give an evaluation of Rubin, the development platform for the dialogue manager.

Key words: dialogue management, flexible natural language tutorial dialogue on proofs, information state update

1 Introduction

In this paper we present and discuss the design and implementation of the dialogue manager for the demonstrator system of the DIALOG project[3] [7,27]. This system was built for demonstration purposes by the DIALOG team[4] at Saarland University for the review of the Collaborative Research Centre 378. 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.

[1] Email: `markb@ags.uni-sb.de`
[2] Email: `chris@ags.uni-sb.de`
[3] http://www.ags.uni-sb.de/~chris/dialog/, http://www.coli.uni-sb.de/sfb/
[4] The DIALOG team is: Christoph Benzmüller, Ivana Kruijff-Korbayova, Manfred Pinkal, Jörg Siekmann, Dimitra Tsovaltzi, Quoc Bao Vo, Magdalena Wolska, Serge Autexier, Armin Fiedler, Erica Melis, Beata Biehl, Mark Buckley, Oliver Culo, Sreedhar Ellisetty, Hussain Syed Sajjad, Marvin Schiller, Andrea Schuh, Jochen Setz, Michael Wirth.

Tutoring should use flexible natural language in order to be effective [25]. A tutorial dialogue is a dialogue whose task 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. 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, including Autotutor [18] and the PACT Geometry Tutor [1] for the physics domain and BEETLE [38], which tutors basic electronics,

To support natural language tutorial dialogue the Dialog project must employ natural language understanding and generation, automated theorem proving, and tutorial reasoning. 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.

The problems which the Dialog project is concentrating on include linguistic analysis of the informal input to the system, evaluation of utterances in terms of soundness, granularity and relevance, and ambiguity resolution at all levels of processing. This means that there must be a tight interplay in the system of the modules responsible for linguistic analysis and automated theorem proving. This is facilitated by a dialogue manager which provides the link between the two modules.

In order to use a traditional automated theorem prover (ATP), the user must be familiar with its logic, since input and output are expressed in this machine-oriented formalism. If an ATP is to be used as a mathematical assistance system, i.e. a mathematician's tool, it must be able to be accessed in the language familiar to the mathematician: a mixture of natural language, formulas and diagrams. For the moment, no ATP fully supports this type of natural interface, but there has been much research in this direction.

One system which abstracts away from the standard interface of ATPs is Proof General [3]. It provides a standard graphical interface for ATPs which is adaptable to the user's expertise level and uses the native proof scripting language of Isabelle/Coq [26]. AUTOMATH [11,12], Mizar [32] and Isabelle/Isar [35] use an approach where input and output are expressed in a formal mathematical language which is both human and machine understandable. Although these languages are human readable, they have a formal style — using keywords and strict structure — which is far from natural language or textbook-style proofs. The *grammatical framework* (GF) [28] attempts to remedy this by allowing the user to define in a λ-calculus a concrete input/output language on top of the abstract syntax of the mathematical expressions. The result is a context-free grammar which can be used both to parse input to and to generate output from the ATP. Although this allows for the definition of a

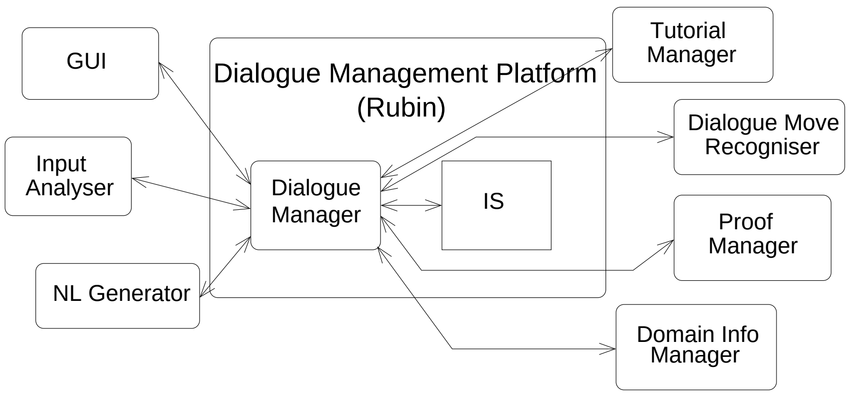


Fig. 1. Architecture of the demonstrator

fragment of natural language, the languages that can be defined are relatively restricted.

Other approaches concentrate solely on a more natural output from the ATP, and leave input in the machine-oriented formalism. The systems Coq [10] and *Theorema* [8] have output components which use schema-based techniques to achieve a pseudo-natural language proof presentation. A further step towards natural langauge output is to apply techniques from the area of natural language generation to produce texts. This approach is used in the presentation component of Nuprl [19], which uses a content planner to select information to be included and a surface realiser to choose words and output sentences. Similarly, PROVERB [20] and its successor *P.rex* [13] use a pipeline architecture of macroplanning, microplanning and surface realisation to generate proofs comparable to those found in mathematical textbooks.

1.1 System Architecture

The Dialog project differs from the approaches outlined above in that it attempts to support a fully flexible natural language interface to a tutorial system on mathematical proofs in both input and output. As such it must interface with subsystems responsible for natural language understanding and generation, automated theorem proving, and tutorial reasoning. We impose no restraints on the language used to communicate with the system in terms of the level of formalism. This goal puts heavy demands on NLU and proof management (the link to the ATP) and on the dialogue manager, which must facilitate their cooperation.

The architecture of the demonstrator is depicted in Figure 1, and shows the following modules connected to the system.

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

Input Analyser This 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.

Dialogue 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 mediates between the system and the Ωmega–Core theorem prover. It evaluates student input and monitors and maintains the proof state.

Domain Info Manager This module determines the mathematical information of the proof step at hand.

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 dialogue moves.

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

In Section 2 we give an outline of the functions of the dialogue manager, followed by a description in Section 3 of the Dialog system modules whose interaction it facilitates. In Section 4 we briefly introduce Rubin, the development platform for the dialogue manager, before presenting the definition of the dialogue manager itself in Section 5. Finally we discuss some results and mention some desiderata for a dialogue manager in the Dialog project.

1.2 Approaches to Dialogue Management

There are a number of candidate approaches to dialogue management which could be suitable for Dialog. Finite-state methods, such as the CSLU toolkit [23], are suited to situations 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 statically encodes 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 [18], 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 have decided to employ is the information state update

(ISU) approach [31]. In this approach the dialogue manager maintains a dialogue context, a description of the state of the dialogue and its participants, which then forms a framework for communication between the external modules associated with the system. The ISU approach has been developed in in the Siridus and Trindi projects, and implemented in TrindiKit [30]. Here the IS is divided into "private" system information, such as internal beliefs of the system, and "public" information shared between the system and the user, such as their common beliefs.

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. It can be changed by update rules which fire based on actions in the world and which update the IS accordingly.

1.3 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 [14]. The task that the student is asked to prove is theorem (1) (where K stands for the complement operation).

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

The examples in this paper 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 manager, tutorial manager and graphical user interface.

2 The Dialogue Manager

The function of the dialogue manager is to maintain a representation of the dialogue context, handle interaction between student and system, and to facilitate communication between system modules. The design of the dialogue manager is based on the information state update approach, and our representation of dialogue context is based on the information state (IS). This is a central data structure storing information about the current state of the dialogue and about the internal states of modules participating in the dialogue.

The dialogue manager is built on Rubin [17], a platform for developing dialogue management applications from CLT company, which is described in Section 4.

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. It does this by coordinating the computations of the system modules, each of which contributes to building the dialogue move with which the system will reply.

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. 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 [33]; it is an adaptation and extension of the DAMSL taxonomy [2]. 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.

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 [36] (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.

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 attempts to determine its correctness, granularity and relevance. 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 output, 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.

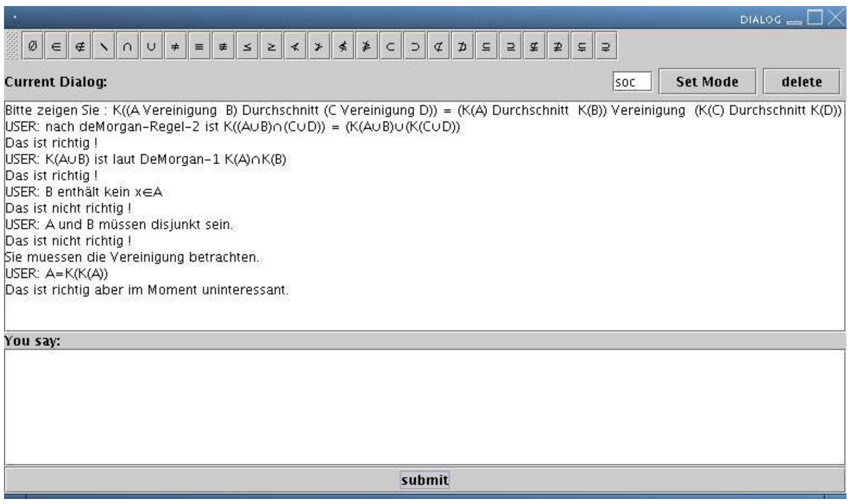
2.2 Inter-module communication

The dialogue manager also acts 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. 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.

3 Modules in the Demonstrator

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



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.

Translation:

SYS:	Please show: K((A∪B)∩(C∪D)) = (K(A)∩K(B))∪(K(C)∩K(D))!
USER:	According to deMorgan-rule-2, it holds that K((A∪B)∩(C∪D)) = (K(A∪B) ∪ K(C∪D))
SYS:	That's correct!
USER:	K(A∪B) is according to DeMorgan-1 K(A) ∩ K(B)
SYS:	That's also correct.
USER:	B does not contain any $x \in A$.
SYS:	That's not correct!
USER:	A and B must be disjoint.
SYS:	That's not correct! You should consider union
USER:	A = K(K(A))
SYS:	That's correct, but not relevant at the moment.

Fig. 2. The DiaWoz tool, extended for the Dialog demonstrator, showing the first 5 moves of the sample dialogue.

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 $K((A \cup B) \cap (C \cup D)) = (K(A \cup B) \cup K(C \cup D))$"; see also Figure 2.

3.1 Graphical User Interface

The GUI of the demonstrator program is an extension of the DiaWoZ tool [14], which has been developed in the Dialog project at the very beginning to support the Wizard-of-Oz experiments in which we collected a corpus of tutorial dialogues. The GUI is presented in Figure 2. In the lower text field the user types his/her input, which when submitted, appears in the upper text field, or conversation field. System utterances also appear in this field.

Input:[5] 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:

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

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 [6], and its linguistic meaning is represented using *Hybrid Logic Dependence Semantics* (HLDS) [5].
Input: The user's utterance in a string (`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 (`LU`, standing for proof Language with Underspecification). This is a language in the spirit of the proof representation language described in [4], but designed for the inter-module communication requirements of the Dialog project.

$$\{ \; \texttt{LM} = @h1(\text{holds} \wedge \text{<}\textsc{criterion}\text{>}(d1 \wedge \text{deMorgan-Regel-2}) \wedge \text{<}\textsc{patient}\text{>}(f1 \wedge \textsc{formula}))$$

```
  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))))
         }
```

We see from the example that the formula and the inference rule (`demorgan-2`) have been successfully determined, whereas the type and direction of the step

[5] The notion of input/output depends 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 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.

remain underspecified.

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 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. `"Address_statement"` means simply that the utterance addresses a preceding statement. The task dimension `"Domain_contribution"` describes a dialogue move which is concerned with resolving the domain task.

3.4 Proof Manager

The proof manager is the mediator between the dialogue manager and the mathematical proof assistant Ωmega–Core [29]. 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. Relevance here refers to whether the proposed step brings the student closer to a complete proof. 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.

The proof manager also tries to resolve ambiguity and underspecification in the representation of the proof step uttered by the student. In doing this it ideally accesses mathematical knowledge stored in MBase [22] and the user model in ActiveMath [24], 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 [4,21,34].

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 [4] (`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.

This representation 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 `LU` of the input analyser.
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 item `StepCat`), and added `ProofCompleted 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: "intersection", hypCon: "union",
  domRel: "", iR: "deMorgan-Regel-2"}
```

Here `domConCat` refers to the categorisation of the proof step from the proof manager. The domain information manager has added `relCon`, referring to the main mathematical concept addressed in the utterance (in this case intersection), and `hypCon`, the hypotaxis of the concept, expressing a kind of interdefinability.

3.6 Tutorial Manager

The role of the tutorial manager [16] is to use pedagogical knowledge to decide on how to give hints to the user. This decision is based on the proof step category (correctness, relevance, 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, also known as hinting strategy. This can be minimal feedback (`"min"`), didactic (`"did"`), in which answers and explanations are constantly provided by the tutor, or socratic (`"soc"`), where hints are used to achieve self-explanation [15].
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= ""})
           }
```

3.7 NL Generator

The natural language generation system used in Dialog is *P.rex*, although for the implementation of the demonstrator, *P.rex*'s function was simulated. *P.rex*, as mentioned in the introduction, is designed to present complete proofs in natural language, and is being adapted for the Dialog project. Adaptation was necessary because in a dialogue setting utterances are produced separately and sequentially, not as a complete text. Also, referents of anaphors are constantly changing as the dialogue model develops.

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 input is the combination of the five dimensions of the dialogue move from the dialogue manager, together with the task dimension and the tutorial mode from the tutorial manager. This example consists mainly of empty fields because the move is simply a confirmation of correctness and has no domain content.

The fields correspond to the dialogue move dimensions given in Section 2.1. 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. 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 modules, and execute dialogue plans. Rubin has been used for instance to support flexible dialogue in an airport flight information system [17].

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 term 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:

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:

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

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:

$$\begin{array}{lcl} \mathit{list} & := & [\ ]\ |\ [\ \mathit{value}\ \{,\ \mathit{value}\}^*\ ] \\ \mathit{struct} & := & \pmb{\{}\mathit{slot}^*\pmb{\}} \end{array}$$

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.

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:

$$\mathit{device\_declaration} \quad := \quad \mathit{device\_name} : \mathit{port\_number}\ ;$$

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. 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 `bool`. They have conditions which are tested for the duration of the plan’s execution so that the plan can

terminate prematurely, for instance if the information the plan seeks has been supplied from another source. Plans can make changes to the IS and make calls to devices.

Input Rules
These are rules which carry out arbitrary actions based on input from devices connected to the dialogue manager. The header of the rule contains its constraints, and has the form

$$\mathit{IS\_constraints}\ \{\_|\mathit{device\_name}\}\ \mathit{input\_pattern}$$

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

$$\{x = 3\}$$

the value of the slot `x` in the information state must be 3 for the rule to fire. The unique name of the device from which the input came must be the same as *device_name*. If "_" is given as the device name, the rule can match input from any device. The *input_pattern* must match[6] 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

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

will only match on input from the device called "IA" 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.

The body of a rule is an inlined plan that can update IS, push other plans and so on. 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

$$\begin{array}{lcl} AS & = & \text{set of all assignments of IS slots} \\ PS & = & \text{set of all possible states of the plan stack} \\ Inputs & = & \text{the set of Rubin data objects} \end{array}$$

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

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 ...)`
`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", ...}`
`complete_proof`	`string`	The complete user proof output by the proof manager when the proof has been completed. `((KEY 1_1)` $\rightarrow$ `((Evaluation (expStepRepr (label 1_1) (formula ...)`
`deleting`	`bool`	A flag which is set to true when the latest user/system turn is to be undone. (`true`/`false`)

Table 1
The information state slots in the dialogue model.

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.

Rubin provides the Java abstract class `Client` which acts as a wrapper linking Rubin with a device. It also provides a graphical user interface which displays the state of the dialogue manager, i.e. the values in the IS, and the input rules which fire. This is useful for prototyping, debugging and testing.

5 The Dialogue Model

The dialogue model for the Dialog demonstrator contains the specification of the information state 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.

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 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 is not yet present.

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 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.

5.1 Input Rules

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

```
_, "Gen", input: {...}
_, "GUI", { student_input = stinput,
            mode = tmode,
            delete = d} : {...}
_, "IA", { 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, {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. }
```

Fig. 3. The input rule for data received from the GUI.

As an example consider the rule for input from the GUI, shown in Figure 3. 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 demonstrator 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`-expression 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 “IA”) 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 =`

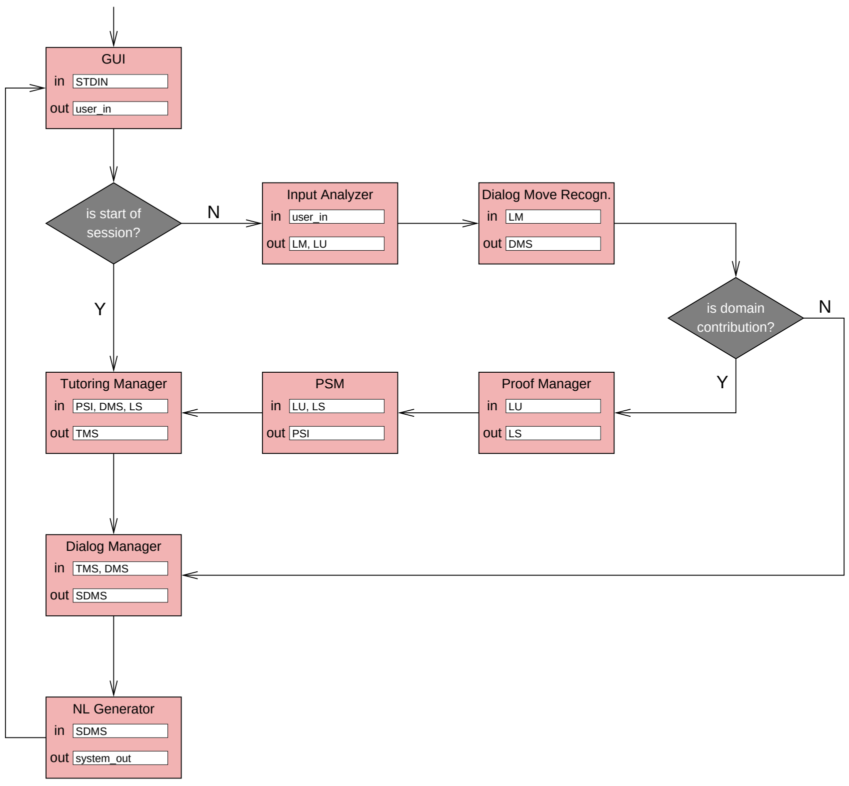


Fig. 4. Information flow in the Dialog demonstrator for a single system turn.

`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 4. 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[7]) and tutorial manager are not called for the system response.

Each arrow in the diagram is actually a transfer of control facilitated by the dialogue manager's communication function. 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 checks if the dialogue move represents a domain contribution, and based on this, passes control to either the proof manager or to itself.

[7] Proof Step Manager, a previous name for the domain information manager.

6 Discussion

6.1 Module Simulation

Because the Dialog demonstrator is at a relatively early stage of development, three modules which are not yet fully implemented had to be simulated: natural language generation, the domain information manager, and the dialogue move recogniser. The natural language 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 task dimension of the dialogue move.

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 five possible types of linguistic meaning. These are: a domain contribution, a request for assistance, and an uninterpretable utterance due to bad grammar, parenthesis mismatch, or a word which is 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 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 input analyser is written in both Java and Perl, and the tutorial manager, proof manager, and the mathematical proof assistant Ωmega–Core are written in LISP.

We solved this using socket communication in a similar way to how the connection between Rubin and its devices works. 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. 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.

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.

6.3 Advantages of 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 some of the software engineering aspects of developing a dialogue application. It provides a GUI in which the information state values can be viewed an altered, and it provides built in data types to represent the content of the information state. It also supports the design of a distributed dialogue application by simplifying module communication with an XML protocol.

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.

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. Indeed, we believe these are true for any dialogue system which integrates many modules for domain and task processing.

Dialog differs from other user interfaces to ATPs in that it aims to support flexible natural language input and output to the tutorial system and therefore to the ATP. This means that a very close 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 flexible adaptable communication between them. For instance, natural language understanding and proof management can ideally use a shared model of the entities which have been mentioned in the dialogue so far to help parsing and proof step evaluation respectively. Maintenance of this model requires tight interleaving which can be facilitated by the dialogue manager.

We now mention some of the requirements of a dialogue manager for the Dialog project which we have identified and to what extent they were supported in the demonstrator.

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 in the dialogue model as an important feature in the future development of the dialogue manager, for instance, the ability to redefine input rules. This could be a 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 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 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. 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 [37]. 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 would be useful in which criteria other than the ordering of rules could be used to choose the most appropriate next rule. It would be possible to inform the dialogue manager using dialogue-level planning, for instance about when to initiate a clarification subdialogue. 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 could 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.

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 execution does not stop. The set of input rules thus forms a chain of invocations, and 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.

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 inflexibility of the input rules is shown in Figure 5. 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. This conditional branching takes place in the input rule for the dialogue move recogniser, even though the decision could have taken place in the input rule for the input analyser, and in this sense the rule for the dialogue move recogniser is not well motivated.

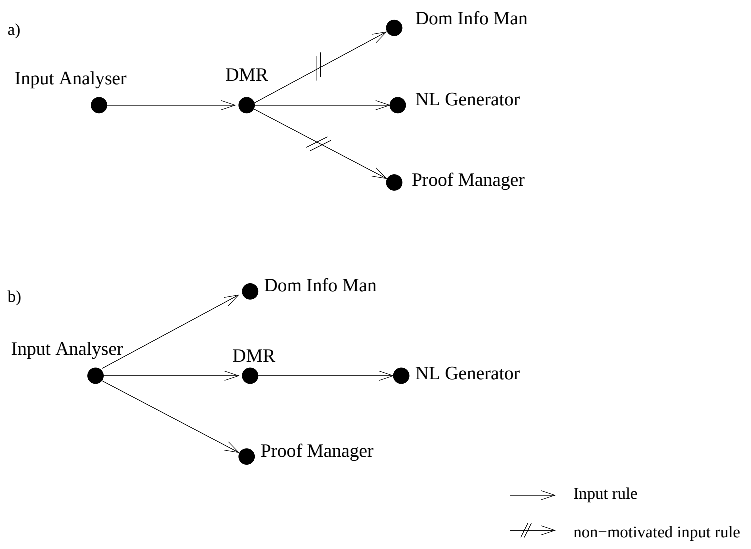


Fig. 5. A section of the information flow, showing (a) scientifically less well motivated input rules and (b) the equivalent well-motivated input rules.

Part (b) shows the scientifically better motivated structure. The results of the input analyser have two effects: a call to the proof manager and a call to the dialogue move recogniser. However 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.

6.5 Summary and Further Work

In this paper we have presented the design and implementation of a dialogue manager based on Rubin for the Dialog demonstrator program. The dialogue manager facilitates a natural language interface to the tutorial system, which in turn accesses the ATP Ωmega–Core, tutorial management and stored mathematical knowledge. 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 and the Rubin platform.

Motivated by the results of the development of the demonstrator, we are investigating the use of agent-based techniques [9] to build a new platform for dialogue management. This should provide the same functionality as Rubin, while also allowing the concurrency and flexibility which we have identified as being necessary for such a dialogue application.

Acknowledgements

This work was supported 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 paper, suggestions and comments on initial drafts, and of course for their contributions to the realisation of the demonstrator system.

References

[1] Aleven, V., K. 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* (1999), pp. 199–206.

[2] Allen, J. and M. Core, *Draft of DAMSL: Dialogue act markup in several layers.*, DRI: Discourse Research Initiative, University of Pennsylvania (1997).

[3] Aspinall, D., *Proof General: A generic tool for proof development*, in: S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science **1785** (2000), pp. 38–42.

[4] Autexier, S., C. Benzmüller, A. Fiedler, H. Horacek and B. Q. Vo, *Assertion-level proof representation with under-specification*, Electronic Notes in Theoretical Computer Science **93** (2003), pp. 5–23.

[5] Baldridge, J. and G.-J. 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.

[6] Baldridge, J. and G.-J. M. Kruijff, *Multi-modal combinatory categorial grammar*, in: *EACL '03: Proceedings of the tenth conference on European chapter of the Association for Computational Linguistics* (2003), pp. 211–218.

[7] Benzmüller, C., A. Fiedler, M. Gabsdil, H. Horacek, I. Kruijff-Korbayova, M. Pinkal, J. Siekmann, D. Tsovaltzi, B. Q. Vo and M. Wolska, *Tutorial dialogs on mathematical proofs*, in: *Proceedings of IJCAI-03 Workshop on Knowledge Representation and Automated Reasoning for E-Learning Systems*, Acapulco, Mexico, 2003, pp. 12–22.

[8] Buchberger, B., *Natural language proofs in nested cells representation*, in: J. Siekmann, F. Pfenning and X. Huang, editors, *Proceedings of the First International Workshop on Proof Transformation and Presentation*, Schloss Dagstuhl, Germany, 1997, pp. 15–16.

[9] Buckley, M., *An Agent-based Platform for Dialogue Management*, in: J. Gervain, editor, *Proceedings of the Tenth ESSLLI Student Session*, Edinburgh, Scotland, 2005, pp. 33–45.

[10] Coscoy, Y., G. Kahn and L. Théry, *Extracting text from proofs*, in: M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications*, number 902 in Lecture Notes in Computer Science (1995), pp. 109–123.

[11] de Bruijn, N. G., *The mathematical language* AUTOMATH, *its usage and some of its extensions*, in: *Symposium on Automatic Demonstration*, number 125 in Lecture Notes in Mathematics (1970), pp. 29–61.

[12] de Bruijn, N. G., *The mathematical vernacular, a language for mathematics with typed sets*, in: R. P. Nederpelt, J. H. Geuvers and R. C. de Vrijer, editors, *Selected Papers on Automath*, Studies in Logic and the Foundations of Mathematics **133**, Elsevier, 1994 pp. 865 – 935.

[13] Fiedler, A., *Dialog-driven adaptation of explanations of proofs*, in: B. Nebel, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)* (2001), pp. 1295–1300.

[14] Fiedler, A. and M. Gabsdil, *Supporting progressive refinement of Wizard-of-Oz experiments*, in: C. P. Rosé and V. Aleven, editors, *Proceedings of the ITS 2002 — Workshop on Empirical Methods for Tutorial Dialogue Systems*, San Sebastián, Spain, 2002, pp. 62–69.

[15] Fiedler, A. and D. Tsovaltzi, *Automating hinting in an intelligent tutorial system*, in: *Proceedings of the IJCAI Workshop on Knowledge Representation and Automated Reasoning for E-Learning Systems*, Acapulco, 2003, pp. 23–35.

[16] Fiedler, A. and D. Tsovaltzi, *Automating hinting in mathematical tutorial dialogue*, in: *Proceedings of the EACL-03 Workshop on Dialogue Systems: Interaction, Adaptation and Styles of Management*, Budapest, 2003, pp. 45–52.

[17] Fliedner, G. and D. 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.

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

[19] Holland-Minkley, A. M., R. Barzilay and R. L. Constable, *Verbalization of high-level formal proofs*, in: *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99) and Eleventh Innovative Application of Artificial Intelligence Conference (IAAI-99)* (1999), pp. 277–284.

[20] Huang, X. and A. Fiedler, *Proof verbalization as an application of NLG*, in: M. E. Pollack, editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI)* (1997), pp. 965–970.

[21] Hübner, M., S. Autexier, C. Benzmüller and A. Meier, *Interactive theorem proving with tasks*, Electronic Notes in Theoretical Computer Science **103** (2004), pp. 161–181.

[22] Kohlhase, M. 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** (2001), pp. 365–402.

[23] McTear, M. F., *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.

[24] Melis, E., 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*, 2001, pp. 385–407.

[25] Moore, J., *What makes human explanations effective?*, in: *Proceedings of the Fifteenth Annual Meeting of the Cognitive Science Society*, Hillsdale, NJ, 1993, pp. 131–136.

[26] Paulson, L. C., "Isabelle: A Generic Theorem Prover," LNCS, Springer Verlag, 1994.

[27] Pinkal, M., J. Siekmann and C. Benzmüller, *Dialog: Tutorial dialog with an assistance system for mathematics* (2004), project report in the Collaborative Research Centre SFB 378 on Resource-adaptive Cognitive Processes.
URL www.ags.uni-sb.de/~chris/papers/R28.pdf

[28] Ranta, A., *Grammatical framework — a type-theoretical grammar formalism*, Journal of Functional Programming **14** (2004), pp. 145–189.

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

[30] Traum, D., J. Bos, R. Cooper, S. Larsson, I. Lewin, C. Matheson and M. Poesio, *A model of dialogue moves and information state revision*, Technical report TRINDI project deliverable D2.1, University of Gothenburg (1999).

[31] Traum, D. and S. 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 .

[32] Trybulec, A. and H. Blair, *Computer Assisted Reasoning with MIZAR*, in: A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI)* (1985), pp. 26–28.

[33] Tsovaltzi, D. and E. Karagjosova, *A view on dialogue move taxonomies for tutorial dialogues*, in: *Proceedings of 5th SIGdial Workshop on Discourse and Dialogue*, Boston, USA, 2004.

[34] Vo, Q. B., C. Benzmüller and S. Autexier, *Assertion application in theorem proving and proof planning*, in: G. Gottlob and T. Walsh, editors, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Acapulco, Mexico, 2003, iSBN 0-127-05661-0.

[35] Wenzel, M., *Isar — a generic interpretative approach to readable formal proof documents*, in: Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin and L. Théry, editors, *Theorem Proving in Higher Order Logics: TPHOLs'99*, Lecture Notes in Computer Science **1690** (1999), pp. 167–184.

[36] Wolska, M., B. Q. 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)* (2004).

[37] Zinn, C., J. D. Moore and M. G. Core, *A 3-tier planning architecture for managing tutorial dialogue*, in: *Proceedings of Intelligent Tutoring Systems, Sixth International Conference*, Biarritz, France, 2002.

[38] Zinn, C., J. D. Moore, M. G. Core, S. Varges and K. 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.