An Agent-Based Architecture for Dialogue Systems*

Mark Buckley and Christoph Benzmüller

Dept. of Computer Science, Saarland University
{markb|chris}@ags.uni-sb.de

Abstract. Research in dialogue systems has been moving towards reusable and adaptable architectures for managing dialogue execution and integrating heterogeneous subsystems. In this paper we present a formalisation of ADMP, an agent-based architecture which supports the development of dialogue applications. It features a central data structure shared between software agents, it allows the integration of external systems, and it includes a meta-level in which heuristic control can be embedded.

1 Introduction

Research in dialogue systems has been moving towards reusable and adaptable architectures for managing dialogue execution and integrating heterogeneous subsystems. In an architecture of this type, different theories of dialogue management can be formalised, compared and evaluated. In this paper we present a formalisation of ADMP[1], an architecture which uses software agents to support the development of dialogue applications. It features a central data structure shared between agents, it allows the integration of external systems, and it includes a meta-level in which heuristic control can be embedded.

We have instantiated the system to support dialogue management. Dialogue management involves maintaining a representation of the state of a dialogue, coordinating and controlling the interplay of subsystems such as domain processing or linguistic analysis, and deciding what content should be expressed next by the system. ADMP applies the information state update (ISU) approach to dialogue management [1]. This approach uses an information state as a representation of the state of the dialogue, as well as update rules, which update the information state as the dialogue progresses. The ISU approach supports the formalisation of different theories of dialogue management.

The framework of our research is the DIALOG project[2], which investigates flexible natural language dialogue in mathematics, with the final goal of natural tutorial dialogue between a student and a mathematical assistance system. In

* This work was supported by the DAAD (German Academic Exchange Service), grant number A/05/05081 and by the DFG (Deutsche Forschungsgemeinschaft), Collaborative Research Centre 378 for Resource-adaptive Cognitive Processes.

[1] The Agent-based Dialogue Management Platform.

[2] http://www.ags.uni-sb.de/dialog/

I. Virbitskaite and A. Voronkov (Eds.): PSI 2006, LNCS 4378, pp. 135–147, 2007.

the course of a tutorial session, a student builds a proof by performing utterances which contain proof steps, thereby extending the current partial proof. The student receives feedback from the DIALOG system after each proof step. This feedback is based on the computations and contribution of numerous systems, such as a domain reasoner or a natural language analysis module. The integration of these modules and the orchestration of their interplay as well as the selection of a next dialogue move which generates the feedback is the task of the dialogue manager.

The work presented in this paper is motivated by an initial prototype dialogue manager for the DIALOG demonstrator [2]. After its development we were able to pinpoint some features which we consider necessary for the DIALOG system, and which the platform presented here supports. The overall design of ADMP is influenced by the design of Ω-Ants [3], a suggestion mechanism which supports interactive theorem proving and proof planning. It uses societies of software agents, a blackboard architecture, and a hierarchical design to achieve concurrency, flexibility and robust distributed search in a theorem proving environment.

Although ADMP has been developed to support dialogue systems, it can be seen as a more general architecture for collaborative tasks which utilise a central data store. For example, we have used ADMP to quickly implement a lean prototype resolution prover for propositional logic.

Our work is related to other frameworks for dialogue management such as TrindiKit, a platform on top of which ISU based dialogue applications can be built. TrindiKit provides an information state, update rules and interfaces to external modules. Another such framework is Dipper [4], which uses an agent paradigm to integrate subsystems.

This paper is structured as follows. In Section 2 we give an overview of the DIALOG project and the role a dialogue manager plays in this scenario. Section 3 outlines the architecture of ADMP. Section 4 presents the formalisation of the system, and Section 5 concludes the paper.

2 The DIALOG Project

The DIALOG project is researching the issues involved in automating the tutoring of mathematical proofs through the medium of flexible natural language. In order to achieve this a number of subproblems must be tackled. An *input analyser* [5] must perform linguistic analysis of utterances. These typically contain both natural language and mathematical expressions and exhibit much ambiguity. In addition to the linguistic analysis the input analyser delivers an underspecified representation of the proof content of the utterance. Domain reasoning is encapsulated in a *proof manager* [6], which replays and stores the status of the student's partial proof. Based on the partial proof, it must analyse the correctness, relevance and granularity of proof steps, and try to resolve ambiguous proof steps. Pedagogical aspects are handled by a *tutorial manager* [7], which decides when and how to give which hints.

These three modules, along with several others such as a natural language generator, collaborate in order to fully analyse student utterances and to compute system utterances. Their computation must be interleaved, since they work with shared information, and this interplay is orchestrated by the dialogue manager. Fig. 1 shows the modules involved in the DIALOG system.

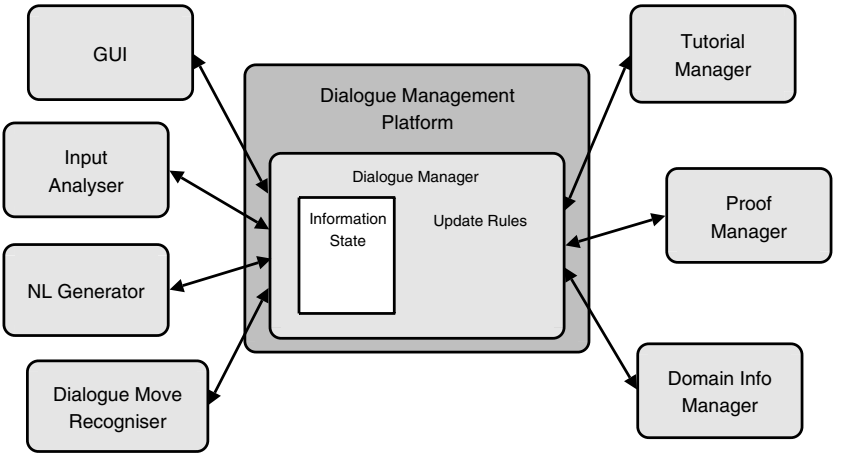

Fig. 1. The DIALOG system

We illustrate how the system works with an example from the DIALOG corpus [8] in Fig. 2, where K stands for the complement operation and U is the universal set. The student has made a correct step (`Stud1`) and thus has begun building a partial proof of the theorem. Later in the proof he makes an irrelevant step (`Stud2`). We now consider how the modules of the system interact to generate the response in (`Sys2`).

The student's utterance first goes to the input analyser, which calculates its linguistic meaning and underspecified proof content. The proof content in this case is a representation of the content of the formula which is given in the utterance. This is then taken up by the proof manager. In the given proof context it assigns the proof step category `correct`, since the formula holds under the assumptions, but also `irrelevant`, because the step does not lead to a proof, or at least does not appear in a proof of the theorem. Simultaneously the *dialogue move recogniser* uses the linguistic meaning to determine the function of the utterance in the dialogue, and encodes this function in a dialogue move [9]. A dialogue move is a multi-dimensional abstraction of the functions of an utterance in a dialogue, such as *question* or *assertion*. In the example the utterance is a *domain contribution* , since it asserts a new piece of information in the theorem proving task.

A *domain information manager* determines what domain information was addressed by the proof step by accessing static mathematical knowledge

`Sys0:` Bitte zeigen Sie: Wenn $A \subseteq K(B)$, dann $B \subseteq K(A)$
Please show that: If $A \subseteq K(B)$ *then* $B \subseteq K(A)$

`Stud1:` $U/A = K(A)$
$U/A = K(A)$

`Sys1:` Das ist richtig! Und weiter?
That's correct! And now?

...

`Stud2:` also ist $A \not\subseteq B$
Therefore it holds that $A \not\subseteq B$

`Sys2:` Nein, das bringt nichts. Wissen Sie, was Sie mit der wenn-dann-Beziehung anfangen müssen?
No, that doesn't help. Do you know what to do with the if-then relation?

Fig. 2. Excerpt from the DIALOG corpus, session soc20k

resources. The tutorial manager uses a combination of these results to add hinting information to the dialogue moves. In this case it decides to explicitly indicate the inappropriateness (*"No"*) and irrelevance (*"that doesn't help"*) of the step. Furthermore, a combination of a student and tutor model result in an explicit hint, namely to draw the student's attention to dissolving the if-then relation which is the head of the theorem.

In general, the result of each module's computation is a contribution of content to some system dialogue move. The final step is that a *natural language generator* generates the utterances constituting the system's response in (`Sys2`) from these dialogue moves. Since a module's computations depend only on information stored in a subset of the information state, their execution order is only partially constrained. This means that many computations can and should take place in parallel, as in the case of the proof manager and dialogue move recogniser in the example above.

DIALOG is an example of a complex system in which the interaction of many non-trivial components takes place. This interaction requires in turn non-trivial control to facilitate the distributed computation which results in the system response. This control function resides in the dialogue manager. As shown in Fig. 1, the dialogue manager forms the hub of the system and mediates all communication between the modules. It furthermore controls the interplay of the modules.

We realised a first DIALOG demonstrator in 2003. It includes a dialogue manager built on top of Rubin [10], a commercial platform for dialogue applications. This dialogue manager integrates each of the modules mentioned above and controls the dialogue. It provides an information state in which data shared between modules is stored, input rules which can update the information state based on input from modules, and interfaces to the system modules.

However, we identified some shortcomings of this first dialogue manager for the demonstrator, and these have formed part of the motivation for the development of ADMP:

– The modules in the system had no direct access to the information state, meaning they could not autonomously take action based on the state of the dialogue.
– The dialogue manager was static, and neither dialogue plans nor the interfaces to modules could be changed at runtime.
– There was also no way to reason about the flow of control in the system.

ADMP solves these problems by using a software agent approach to information state updates and by introducing a meta-level. The meta-level is used to reason about what updates should be made, and provides a place where the execution of the dialogue manager can be guided.

3 Architecture

The central concepts in the architecture of ADMP are information states and update rules, and these form the core of the system. An information state consists of slots which store values, and can be seen as an attribute-value matrix. It is a description of the state of the dialogue at a point in time, and can include information such as a history of utterances and dialogue move, the results of speech recognition or a representation of the beliefs of dialogue participants. Update rules encode transitions between information states, and are defined by a set of preconditions, a list of sideconditions, and a set of effects. Preconditions constrain what information states satisfy the rule, sideconditions allow arbitrary functions to be called within the rule, and effects describe the changes that should be made to the information state in order to carry out the transition that the rule encodes.

An update rule is embodied by an update rule agent, which carries out the computation of the transition that the update rule encodes. These check if the current information state satisfies the preconditions of the rule. When this is the case, they compute an *information state update* representing the fully instantiated transition. An information state update is a mapping from slotnames in the information state to the new values they have after the update is executed. We introduce information state updates as explicit objects in ADMP in order to be able to reason about their form and content at the meta-level.

As an example, we consider the information state in (1), a subset of the information state of the DIALOG system[3]. Here the user's utterance is already present in the slot `user_utterance`, but the linguistic meaning in the slot `lm` has not yet been computed. The slot `lu` stores a representation of the proof content of the utterance, and `eval_lu` stores its evaluated representation.

(1)
$$\text{IS} \begin{bmatrix} \texttt{user\_utterance} & \text{"also ist } A \not\subseteq B\text{"} \\ \texttt{lm} & \text{""} \\ \texttt{lu} & \text{""} \\ \texttt{eval\_lu} & \text{""} \end{bmatrix}$$

[3] In general an information state will contain richer data structures such as XML objects, but for presentation we restrict ourselves here to strings.

The update rule in (2) represents transitions from information states with a non-empty `user_utterance` slot to information states in which the `lm` and `lu` slots have been filled with the appropriate values.

$$(2)\quad \frac{\{\texttt{non\_empty(user\_utterance)}\}}{\{\texttt{lm} \rightarrow \texttt{p}\ ,\ \texttt{lu} \rightarrow \texttt{q}\}} \quad \begin{array}{l} < \texttt{r} := \texttt{input\_analyser(user\_utterance)}, \\ \quad \texttt{p} := \textit{extract\_lm}(\texttt{r}), \\ \quad \texttt{q} := \textit{extract\_lu}(\texttt{r}) > \end{array}$$

In ADMP's update rule syntax this rule is defined as:

```
(3) (ur~define-update-rule
      :name "Sentence Analyser"
      :preconds ((user_utterance :test #'ne-string))
      :sideconds ((r :function input_analyser
                     :slotargs (user_utterance))
                  (p :function extract-lm :varargs (r))
                  (q :function extract-lu :varargs (r))
                  )
      :effects ((lm p) (lu q))
    )
```

The precondition states that the slot `user_utterance` must contain a non-empty string. When this is the case, the rule can fire. It carries out its sideconditions, thereby calling the function `input_analyser`, which performs the actual computation and calls the module responsible for the linguistic analysis of utterances. Rule (2) thus represents the input analyser. The result of this computation is an object containing both the linguistic meaning of the utterance and an underspecified representation of the proof content. The functions *extract_lm* and *extract_lu* access the two parts and store them in the variables `p` and `q`, respectively. The information state update that the rule computes maps the slot name `lm` to the linguistic meaning of the utterance and the slot name `lu` to its proof content.

Rule (4) represents the proof manager, and picks up the proof content of the utterance in the slot `lu`.

$$(4)\quad \frac{\{\texttt{non\_empty(lu)}\}}{\{\texttt{eval\_lu} \rightarrow \texttt{r}\}} \quad < \texttt{r} := \texttt{pm\_analyse(lu)} >$$

The proof manager augments the information in `lu` by attempting to resolve underspecification and assign correctness and relevance categories, and the resulting update maps `eval_lu` to this evaluated proof step. A similar update rule forms the interface to the dialogue move recogniser, which uses the linguistic meaning of the utterance in `lm` to compute the dialogue move it represents. Since these two computations are both made possible by the result of the update from the input analyser, they can run in parallel.

Fig. 3 shows the architecture of ADMP. On the left is the information state. Update rules have in their preconditions constraints on some subset of the information state slots and are embodied by update rule agents, which are shown here next to the information state. When an update rule agent sees that the preconditions of its rule hold, the rule is applicable and can fire. The agent then executes each of the sideconditions of the rule, and subsequently computes the

information state update that is expressed by the rule's effects. The resulting information state update is written to the update blackboard, shown in the middle of the diagram.

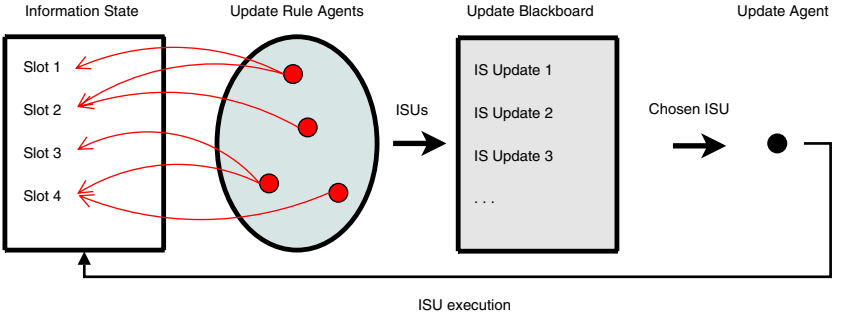

Fig. 3. The architecture of ADMP

The update blackboard collects the proposed updates from the update rule agents. These agents act in a concurrent fashion, so that many of them may be simultaneously computing results; some may return results quickly and some may perform expensive computations, e.g. those calling external modules. Thus the set of entries on the update blackboard can grow continually. On the far right of the diagram is the update agent, which surveys the update blackboard. After a timeout or some stimulus it chooses the heuristically preferred update (or a combination of updates) and executes it on the current information state. This completes a transition from one information state to the next.

Finally the update agent resets the update rule agents. Agents for whom the content of the slots in their preconditions has not changed can continue to execute since they will then be computing under essentially the same conditions (i.e. the information that is relevant to them is the same). Agents for whom the slots in the preconditions have changed must be interrupted, even if their preconditions still happen to hold. This is because they are no longer computing within the correct current information state.

4 A Formal Specification of ADMP

We now give a concise and mathematically rigorous specification of ADMP. We introduce the concepts and terminology necessary to guarantee the well-definedness of information states and update rules, and we give an algorithmic description of the update rule agents and the update agent.

Information States and Information State Updates. First, we fix some data structures for the slot names and the slot values of an information state. In our scenario it is sufficient to work with strings in both cases (alternatively we could work with more complex data structures). Let $\mathcal{A}$ and $\mathcal{B}$ be alphabets.

We define the *language for slot names* as $SlotId := \mathcal{A}^*$ and the *language for slot values* as $SlotVal := \mathcal{B}^*$. In our framework we want to support the checking of certain properties for the values of single slots. Thus we introduce the notion of a Boolean test function for slot values. A *Boolean test function* is a function $f \in \mathcal{BT} := SlotVal \to \{\top, \bot\}$.

Next, we define information state slots as triples consisting of a slot name, a slot value, and an associated Boolean test function. The set of all possible *information state slots* is $Slots := SlotId \times \mathcal{BT} \times SlotVal$. Given an information state slot $u = (s, b, v)$, the slot name, the test function, and the slot value can be accessed by the following projection functions: $slotname(u) := s$, $slotfunc(u) := b$ and $slotval(u) := v$.

Information states are sets of information state slots which fulfil some additional conditions. Given $r \subseteq Slots$, we call r a *valid information state* if $r \neq \emptyset$ and for all $u_1, u_2 \in r$ we have $slotname(u_1) = slotname(u_2) \Rightarrow u_1 = u_2$. We define $\mathcal{IS} \subset \mathcal{P}(Slots)$ to be the set of all valid information states. The set of all slot names of a given information state $r \in \mathcal{IS}$ can be accessed by a function $slotnames : \mathcal{IS} \to \mathcal{P}(SlotId)$ which is defined as follows

$$slotnames(r) = \{s \in SlotId \mid \exists\, u \in r \,.\, slotname(u) = s\}$$

We define a function $read : \mathcal{IS} \times SlotId \to SlotVal$ to access the value of a slot in an information state where $read(r, s) = slotval(u)$ for the unique $u \in r$ with $slotname(u) = s$.

In our framework information states are dynamically updated, i.e. the values of information state slots are replaced by new values. Such an *information state update* is a mapping from slots to their new values. The set of all valid information state updates μ is denoted by $\mathcal{ISU}$, the largest subset of $\mathcal{P}(SlotId \times SlotVal)$ for which the following restriction holds: $\forall (s_1, v_1), (s_2, v_2) \in \mu \,.\, s_1 = s_2 \Rightarrow v_1 = v_2$ for all $\mu \in \mathcal{ISU}$. We define $\mathcal{ISU}_\bot := \mathcal{ISU} \cup \{\bot\}$. An information state update $\mu \in \mathcal{ISU}$ is *executable* in an information state $r \in \mathcal{IS}$ if the slot names addressed in μ actually occur in r and if the new slot values suggested in μ fulfil the respective Boolean test functions, i.e.

$$executable(r, \mu) \text{ iff } \forall (s, v) \in \mu \,.\, \exists\, u \in r \,.\, slotname(u) = s \wedge slotfunc(u)(v) = \top$$

We overload the function *slotnames* from above and analogously define it for information state updates. Information state updates are executed by a function $execute\_update : \mathcal{IS} \times \mathcal{ISU} \to \mathcal{IS}$. Given an information state $r \in \mathcal{IS}$ and an information state update $\mu \in \mathcal{ISU}$ we define

$$execute\_update(r, \mu) = \begin{cases} r & \text{if not } executable(r, \mu) \\ r^- \cup r^+ & \text{otherwise} \end{cases}$$

where

$$r^- := (r \setminus \{(s, b, v) \in r | s \in slotnames(\mu)\}$$
$$r^+ := \{(s', b', v') \mid (s', v') \in \mu \wedge \exists u \in r \,.\, s' = slotname(u) \wedge b' = slotfunc(u)\}$$

Update Rules. Update rules use the information provided in an information state to compute potential information state updates. They consist of preconditions, sideconditions and effects.

The preconditions of an update rule identify the information state slots that the rule accesses information from. For each identified slot an additional test function is provided which specifies an applicability criterion. Intermediate computations based on information in the preconditions are performed by the sideconditions of the update rules. For this, a sidecondition may call complex external modules, such as the linguistic analyser or the domain reasoner. The results of these side-computations are bound to variables in order for them to be accessible to subsequent sideconditions and to pass them over from the sideconditions to the effects of a rule. We now give a formal definition of each part in turn.

Let $s \in \mathit{SlotId}$ and $b \in \mathcal{BT}$. The tuple (s, b) is called an *update rule precondition*. The set of all update rule preconditions is denoted by $\mathcal{C} := \mathit{SlotId} \times \mathcal{BT}$. We define projection functions $\mathit{pc\_slotname} : \mathcal{C} \rightarrow \mathit{SlotId}$ and $\mathit{pc\_testfunc} : \mathcal{C} \rightarrow \mathcal{BT}$ such that $\mathit{pc\_slotname}(pc) = s$ and $\mathit{pc\_testfunc}(pc) = b$ for all $pc = (s, b)$. An information state $r \in \mathcal{IS}$ *satisfies* an update rule precondition $pc = (s, b)$ if the function b applied to the value of the slot in r named s returns $\top$, i.e. $\mathit{satisfies}(r, pc)$ iff $\exists u \in r \,.\, \mathit{pc\_testfunc}(pc)(\mathit{slotval}(u)) = \top \wedge \mathit{slotname}(u) = \mathit{pc\_slotname}(pc)$. We overload the predicate *satisfies* and define it for sets of preconditions $\mathcal{C}' \subseteq \mathcal{C}$ and information states $r \in \mathcal{IS}$ as follows: $\mathit{satisfies}(r, \mathcal{C}')$ holds if each precondition in $\mathcal{C}'$ is satisfied by r.

Let $v \in \mathit{Var}$ be a variable where Var is a set of variables distinct from the languages $\mathcal{A}^*$ and $\mathcal{B}^*$, let $(v_1 \ldots v_m) \in \mathit{Var}^m$ be an m-tuple of variables, let $(s_1 \ldots s_n) \in \mathit{SlotId}^n$ be an n-tuple of slot names, and let $f : \mathit{SlotVal}^n \rightarrow \mathit{SlotVal}^m \rightarrow \mathit{SlotVal}$ be a function[4] $(n, m \geq 0)$. A *single sidecondition* is thus given by the quadruple $(v, (s_1, \ldots, s_n), (v_1, \ldots, v_m), f)$. The set of all single sideconditions is denoted by $\mathcal{D} := \mathit{Var} \times \mathit{SlotId}^n \times \mathit{Var}^m \times (\mathit{SlotVal}^n \rightarrow \mathit{SlotVal}^m \rightarrow \mathit{SlotVal})$.

Given the set $\mathcal{D}$ of single sideconditions sc_i, the sideconditions of an update rule are now modelled as lists $l := <sc_1, \ldots, sc_n>$, $n \geq 0$. We further provide projection functions $\mathit{sc\_var} : \mathcal{D} \rightarrow \mathit{Var}$, $\mathit{sc\_slottuple} : \mathcal{D} \rightarrow \mathit{SlotId}^n$, $\mathit{sc\_slotnames} : \mathcal{D} \rightarrow \mathcal{P}(\mathit{SlotId})$, $\mathit{sc\_vartuple} : \mathcal{D} \rightarrow \mathit{Var}^m$, $\mathit{sc\_varnames} : \mathcal{D} \rightarrow \mathcal{P}(\mathit{Var})$ and $\mathit{sc\_func} : \mathcal{D} \rightarrow (\mathit{SlotVal}^n \rightarrow \mathit{SlotVal}^m \rightarrow \mathit{SlotVal})$, such that for all $sc = (v, (s_1, \ldots, s_n), (v_1, \ldots, v_m), f) \in \mathcal{D}$ it holds that $\mathit{sc\_var}(sc) = v$, $\mathit{sc\_slottuple}(sc) = (s_1, \ldots, s_n)$, $\mathit{sc\_slotnames}(sc) = \{s_1, \ldots, s_n\}$, $\mathit{sc\_vartuple}(sc) = (v_1, \ldots, v_m)$, $\mathit{sc\_varnames}(sc) = \{v_1, \ldots, v_m\}$ and $\mathit{sc\_func}(sc) = f$.

A *sidecondition list* l is called valid if two conditions hold: for all $sc_i, sc_j \in l$ with $i \neq j$ we must have $\mathit{sc\_var}(sc_i) \neq \mathit{sc\_var}(sc_j)$ and for all $sc_i \in l$ we must have $\mathit{sc\_varnames}(sc_i) \subseteq \{v | \exists\ sc_j \in l \,.\, j < i \wedge v = \mathit{sc\_var}(sc_j)\}$. The set of all valid sidecondition lists is denoted as $\mathcal{D}_l$.

Let $s \in \mathit{SlotId}$ and $v \in \mathit{Var}$ be a variable. The tuple (s, v) is called an *update rule effect*. The set of all update rule effects is denoted by $\mathcal{E} := \mathit{SlotId} \times \mathit{Var}$.

[4] We assume the right-associativity of $\rightarrow$.

We provide projection functions $e\_slotname : \mathcal{E} \rightarrow SlotId$ and $e\_var : \mathcal{E} \rightarrow Var$ such that $e\_slotname((s, v)) = s$ and $e\_var((s, v)) = v$.

Let $\mathcal{U}$ be a set of rule names (distinct from $\mathcal{A}^*$, $\mathcal{B}^*$, and Var). An *update rule* is a quadruple $\nu \in \mathcal{UR} := \mathcal{U} \times \mathcal{P}(\mathcal{C}) \times \mathcal{D}_l \times \mathcal{P}(\mathcal{E})$. An update rule $\nu = (n, c, d, e) \in \mathcal{UR}$ is *well-defined* w.r.t. the information state r if

1. the slotnames mentioned in the preconditions actually occur in r, i.e, for all $pc \in c$ we have $pc\_slotname(pc) \in slotnames(r)$,
2. each slot that is accessed by a sidecondition function has been mentioned in the preconditions, i.e., $(\bigcup_{d_i \in d} sc\_slotnames(d_i)) \subseteq \{s \in SlotId \mid \exists\ pc \in c\ .\ pc\_slotnames(pc) = s\}$,
3. the variables occurring in the effects have been initialised in the sideconditions, i.e., $\{v \in Var \mid \exists\ e_i \in e\ .\ e\_var(e_i) = v\} \subseteq \{v \in Var \mid \exists\ sc \in d\ .\ sc\_var(sc) = v\}$, and
4. the slotnames in the effects refer to existing slots in the information state r, i.e., $\{s \in SlotId \mid \exists e_i \in e\ .\ e\_slotname(e_i) = s\} \subseteq slotnames(r)$.

Let $\nu = (n, c, d, e) \in \mathcal{UR}$ be an update rule and $r \in \mathcal{IS}$ be an information state. ν is called *applicable* in r if ν is well-defined w.r.t. r and $satisfies(r, c)$ holds. This is denoted by $applicable(r, \nu)$.

Update Rule Agents. Update rule (software) agents encapsulate the update rules, and their task is to compute potential information state updates. The suggested updates are not immediately executed but rather they are passed to an update blackboard for heuristic selection. Update rule agents may perform their computations in a distributed fashion.

An update rule agent embodies a function $execute\_ur\_agent : \mathcal{UR} \rightarrow (\mathcal{IS} \rightarrow \mathcal{ISU}_\perp)$. The function $execute\_ur\_agent(\nu)$ takes an update rule ν and returns a function (lambda term) representing the computation that that rule defines. The new function can then be applied to a given information state in order to compute a suggestion for how to update this information state. For each update rule we obtain a different software agent.

We introduce a macro `sc_evaluate` which abbreviates the retrieval of the values in the variables and slotnames in the body of sidecondition and the computation of the value which is to be stored in the sidecondition's variable. We use **function_call** to apply a function to the arguments which follow it and **value_of** to retrieve the value stored in a variable.

$$
\begin{array}{l}
\texttt{sc\_evaluate}(sc)\ = \\
\quad \textbf{let}\ (s_1, \ldots, s_n) := sc\_slottuple(sc) \\
\quad\quad \textbf{let}\ (v_1, \ldots, v_m) := sc\_vartuple(sc) \\
\quad\quad\quad \textbf{let}\ (t_1, \ldots, t_m) := (\textbf{value\_of}(v_1), \ldots, \textbf{value\_of}(v_m)) \\
\quad\quad\quad\quad \textbf{function\_call}(sc\_func(sc), (read(r, s_1), \ldots, read(r, s_n)), (t_1, \ldots, t_m))
\end{array}
$$

We now define *execute_ur_agent* as

$$
\begin{array}{l}
\mathit{execute\_ur\_agent}(\nu = (n, c, d, e)) = \\
\quad \lambda r \;.\; \textbf{if } \mathit{applicable}(r, \nu) \\
\qquad \textbf{then} \\
\qquad\quad \textbf{let } <sc_1, \ldots, sc_n> := d \\
\qquad\qquad \textbf{let } \mathit{sc\_var}(sc_1) := \texttt{sc\_evaluate}(sc_1) \\
\qquad\qquad\quad \textbf{let } \mathit{sc\_var}(sc_2) := \texttt{sc\_evaluate}(sc_2) \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad \textbf{let } \mathit{sc\_var}(sc_n) := \texttt{sc\_evaluate}(sc_n) \\
\qquad\qquad\qquad\quad \{(s, v) | \exists (s, \mathit{sc\_var}(sc_i)) \in e \;.\; v = \textbf{value\_of}(\mathit{sc\_var}(sc_i))\} \\
\qquad \textbf{else } \bot
\end{array}
$$

Update Blackboard and Update Agent. An *update blackboard* is modelled as a set of information state updates $w \in \mathcal{UB} := \mathcal{P}(\mathcal{ISU})$, and stores proposed updates to the current information state. The *update agent* investigates the entries on the update blackboard, heuristically chooses one of the proposed information state updates and executes it. We assume a user-definable function $\mathit{choose} : \mathcal{UB} \to \mathcal{ISU}$ which realises the heuristic choice based on some heuristic ordering criterion $>_{\mathcal{UB}} : \mathcal{ISU} \times \mathcal{ISU}$. A simple example of a partial ordering criterion $>_{\mathcal{UB}}$ is

$$\mu_1 >_{\mathcal{UB}} \mu_2 \;\text{ iff }\; \mathit{slotnames}(\mu_2) \subseteq \mathit{slotnames}(\mu_1)$$

In fact, *choose* may be composed of several such criteria, and clearly the overall behaviour of the system is crucially influenced by them. The update agent now embodies a function $\mathit{update\_agent} : \mathcal{UB} \times (\mathcal{UB} \to \mathcal{ISU}) \times \mathcal{IS} \to \mathcal{IS}$ which is defined as

$$\mathit{update\_agent}(w, \mathit{choose}, r) = \mathit{execute\_update}(r, \mathit{choose}(w))$$

5 Conclusion

In this paper we have presented a formalisation of ADMP, a platform for developing dialogue managers using the information state update approach. We were motivated by the need to integrate many complex and heterogeneous modules in a flexible way in a dialogue system for mathematical tutoring. These modules must be able to communicate and share information with one another as well as to perform computations in parallel.

ADMP supports these features by using a hierarchical agent-based design. The reactive nature of the update rule agents allows for the autonomous concurrent execution of modules triggered by information in the information state. This furthermore obviates the need for a strict pipeline-type control algorithm often seen in dialogue systems, since agents can execute without being explicitly called. Interfacing the dialogue manager with system modules is also simplified by using

the agent paradigm, because adding a new module involves only declaring a new update rule. Finally, the meta-level provides a place where overall control can take place if needed.

ADMP thus allows the formalisation of theories of dialogue in the information state update approach, offering the functionality of related systems like TrindiKit and Dipper. However by introducing an explicit heuristic layer for overall control it allows reasoning about the execution of the dialogue manager which these two systems do not support.

An instantiation of ADMP is achieved by declaring an information state, a set of update rules which operate on the information state, and a *choose* function, whereby a developer can fall back to a default function such as suggested in the previous section. A user-defined *choose* function should compute valid $\mathcal{ISU}$s, also in the case where $\mathcal{ISU}$s from the update blackboard are merged. As an example, a conservative merge strategy would simply reject the merging of pairs of $\mathcal{ISU}$s whose slotname sets intersect. Update rule agents and the update agent are automatically generated from the update rule declarations.

We have recently implemented ADMP and given an instantiation for the DIALOG system which uses eleven update rules and requires no declaration of control structure. We have also shown that we can implement a propositional resolution prover in ADMP with four agents and five information state slots, which corresponds to just 40 lines of code. Extensions such as a set of support strategy can be realised simply by adding agents, possibly at runtime.

We foresee as future work the extension of our agent concept to include for instance resource sensitivity, and the investigation of further default heuristics for the dialogue scenario. Other interesting work is to turn the specification given in this paper into a formalisation within a higher-order proof assistant such as ISABELLE/HOL, HOL or OMEGA and to verify its properties.

References

1. Traum, D., Larsson, S.: The information state approach to dialogue management. In van Kuppevelt, J., Smith, R., eds.: Current and new directions in discourse and dialogue. Kluwer (2003)
2. Buckley, M., Benzmüller, C.: A Dialogue Manager supporting Natural Language Tutorial Dialogue on Proofs. Electronic Notes in Theoretical Computer Science (2006) To appear.
3. Benzmüller, C., Sorge, V.: Ω-Ants – An open approach at combining Interactive and Automated Theorem Proving. In Kerber, M., Kohlhase, M., eds.: 8th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus-2000), AK Peters (2000)
4. Bos, J., Klein, E., Lemon, O., Oka, T.: Dipper: Description and formalisation of an information-state update dialogue system architecture. In: Proceedings of the 4th SIGdial Workshop on Discourse and Dialogue, Sapporo, Japan (2003)
5. Horacek, H., Wolska, M.: Interpreting Semi-Formal Utterances in Dialogs about Mathematical Proofs. Data and Knowledge Engineering Journal **58**(1) (2006) 90–106

6. Benzmüller, C., Vo, Q.: Mathematical domain reasoning tasks in natural language tutorial dialog on proofs. In Veloso, M., Kambhampati, S., eds.: Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05), Pittsburgh, Pennsylvania, USA, AAAI Press / The MIT Press (2005) 516–522
7. Tsovaltzi, D., Fiedler, A., Horacek, H.: A Multi-dimensional Taxonomy for Automating Hinting. In Lester, J.C., Vicari, R.M., Paraguaçu, F., eds.: Intelligent Tutoring Systems, 7th International Conference (ITS 2004). Number 3220 in LNCS, Springer (2004) 772–781
8. Benzmüller, C., Fiedler, A., Gabsdil, M., Horacek, H., Kruijff-Korbayová, I., Pinkal, M., Siekmann, J., Tsovaltzi, D., Vo, B.Q., Wolska, M.: A Wizard-of-Oz experiment for tutorial dialogues in mathematics. In: Proceedings of the AIED Workshop on Advanced Technologies for Mathematics Education, Sydney, Australia (2003) 471–481
9. Allen, J., Core, M.: Draft of DAMSL: Dialogue act markup in several layers. DRI: Discourse Research Initiative, University of Pennsylvania (1997)
10. Fliedner, G., Bobbert, D.: 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)