Intelligent Information Presentation for Tutoring Systems

CLAUS ZINN, JOHANNA D. MOORE, AND MARK G. CORE

INTELLIGENT INFORMATION PRESENTATION FOR TUTORING SYSTEMS

School of Informatics, University of Edinburgh, 2 Buccleuch Place, Edinburgh EH8 9LW, UK

Abstract. Effective human tutoring has been compared to a delicate balancing act. Students must be allowed to discover and correct problems on their own, but the tutor must intervene before the student becomes frustrated or confused. Natural language dialogue offers the tutor many ways to lead the student through a line of reasoning, and to indirectly notify the student of an error and use a series of hints and followup questions to get the student back on track. These sequences typically unfold across several conversational turns, during which the student can make more errors, initiate topic changes, or give more information than requested. Thus to support tutorial interactions, we require an intelligent information presentation system that can plan ahead, but is able to adapt its plan to the dynamically changing situation. In this paper we discuss how we have adapted the three-layer architecture developed by researchers in robotics to the management of tutorial dialogue.

1. INTRODUCTION

Much of the work on intelligent information presentation has focused on systems that support information-seeking (*e.g.*, providing flight times and fares), assist decision-making (*e.g.*, comparison shopping, logistics planning), or describe objects and artefacts (*e.g.*, museum guides). In such applications, while we expect that the system has more information than the user, we also assume that the user understands the domain of discourse and is able to use the information the system provides in order to choose an option, or make a decision, or assimilate the information they receive. In this paper, we focus on intelligent information presentation in the domain of tutoring, where the system is trying to teach the user new concepts and correct user misconceptions. To motivate our approach, we describe the unique characteristics of human tutorial interaction, and present an architecture for intelligent information presentation for tutorial applications.

1.1 Intelligent Information Presentation for Tutoring

Studies show that one-to-one human tutoring is more effective than other modes of instruction. A meta-analysis of the findings from 65 independent evaluations of school tutoring programs found that tutoring raised students' performance by 0.40 standard deviations (Cohen, Kulik, & Kulik, 1982). Results with good tutors are even more promising. For example, the average student who received one-to-one tutoring with a good tutor scored 2.0 standard deviations above the average student

T.H.E. Editor(s) (ed.), Book title, 1—6.

who received standard classroom instruction, and 1.0 standard deviation above students in a mastery learning condition (Bloom, 1984).
From its inception, the goal of research in computer-based tutoring environments has been to model the effective behaviors of good human tutors, and in so doing to create an optimal educational tool. There is mounting evidence from cognitive psychology that important (or what many call "deep") learning is most likely to occur when students encounter obstacles and work around them, and explain to themselves what worked and what did not, and how new information fits in with what they already know (Chi, Bassok, Lewis, Reimann, & Glaser, 1989; Chi, de Leeuw, Chiu, & LaVancher, 1994; Ohlsson, Rees, 1991; VanLehn, 1990). This is consistent with the constructivist movement in education, which argues that students learn best when they are active participants in the learning process and construct knowledge for themselves.

Debates about what makes human tutoring effective, and how this might be captured in a computer-based learning environment, led to several detailed studies of human tutoring (Lepper & Chabay, 1988; McArthur, Stasz, & Zmuidzinas, 1990; Merrill, Reiser, & Landes (1992); Fox, 1993; Graesser & Person, 1994). The consensus from these studies is that experienced human tutors maintain a delicate balance, allowing students to do as much of the work as possible and to maintain a feeling of control, while providing students with enough guidance to keep them from becoming frustrated or confused. Maintaining this delicate balance requires that a tutor be flexible. Our and others' analyses of human tutorial interactions show that human tutors use a variety of strategies, including hinting (Hume, Michael, Rovick, & Evens, 1996), drawing students' attention to an error (often indirectly) and providing students an opportunity for repair (Fox, 1993; Lepper & Chabay, 1988), pointing out features of the solution that are incorrect, scaffolding (Chi, Siler, Jeong, Yamauchi, & Hausmann, 2001), and so on. In addition, human tutors strategically moderate their feedback. They sometimes intervene immediately after an error has occurred, but at other times allow the student to proceed with the solution, returning to the error later (Littman, Pinto, & Soloway, 1990). (Merrill, Reiser, Ranney, & Trafton, 1992) argue that human tutorial guidance appears to be structured around impasses, and the content and timing of feedback are dependent on the error or impasse encountered.

Human tutoring is a collaborative process, in which tutor and student work together to repair errors. It is a highly interactive process, with the tutor providing constant feedback to support students' problem solving. (Merrill, Reiser, Ranney, & Trafton, 1992) argue that regardless of the timing or content of the intervention, human tutors carefully design their feedback to allow students to do as much of the work as possible, while still preventing floundering. (Fox, 1993, p. 122) observes that "the tutor and student both make use of strategies which maximize the student's opportunity to correct his/her own mistake." In addition, tutors avoid directly telling the student that they are wrong or precisely how a step is incorrect. Instead tutors indirectly guide students through the process of error detection and correction.

Human tutors interact with students via natural language dialogue, sometimes including equations or references to diagrams or simulated models of the domain. They prompt students to construct knowledge, give explanations, assess the student's understanding, and so on, all via natural language. They get and give linguistic cues about how the dialogue is progressing. These cues give tutors information about the student's understanding of the material, and allow the tutor to determine when a strategy is working or when another tactic is needed.

Natural language dialogue is an ideal medium for this type of interaction because it offers many indirect techniques for notifying students that a step in the solution requires repair. (Fox, 1993) found that tutors provide frequent feedback indicating that a step is okay. A short hesitation in responding "okay" typically led the student to assume that something was amiss with the current step, and frequently led students to repair their own errors. When more explicit help was required, the tutor focused the student's attention on the part of the solution that required modification or on information that was useful for repairing the error. Although students sometimes explicitly request guidance or affirmation that their step is correct, this usually is not necessary because the tutor provides such information through hints, leading questions, verbal agreement, and other indirect methods.

1.2 Managing Tutorial Dialogue

Sect. 1.1 described the benefits of natural language dialogue as a presentation modality for tutoring. In this section, we describe the requirements that must be met to build such a system for intelligent information presentation:

- Presentations must unfold over many conversational turns, even when it would be possible to present all of the information in a single contribution. This is crucial, because the system must give the student opportunities to contribute to the solutions and must not ignore student's signs of confusion.

- The tutor system must have the ability to ask students questions that it "knows" the answer to, either to prompt the student to provide the information to facilitate knowledge construction, or to diagnose the level of the student's knowledge.

- The tutor must understand student utterances well enough to respond appropriately.

- The tutor system must have the ability to react to unexpected events. By evaluating the current dialogue situation, it must be able to revise its current plan or postpone the refinement of a sketchy plan until the situation provides the necessary information. In particular, the tutor is required

    1. not to ignore student confusion,

2. to encourage the student to recognise and correct their own errors,
3. to abandon questions that are no longer relevant,
4. to handle multiple student actions in a single turn, and
5. to deal with student-initiated topic changes.

These five sub-requirements stress the need for a tutorial agent to monitor the execution of its dialogue strategies. In the case of failure, the agent needs to adapt its plan to the new situation: inserting a plan for a sub-dialogue to handle student confusion or a student misconception (1+2), deleting parts of a dialogue plan because their effects are now irrelevant or already achieved (3+4), or reorganising sub-plans to handle topic changes (5).

Consequently, there is no need for tutorial dialogue managers to generate elaborate discourse plans in advance. Given the dynamics of tutorial dialogue ─ the large number of potential student actions at any point and the limited ability of the tutor to predict them – it is a more viable approach to enter a tutorial conversation with a sketchy high-level dialogue plan. As the dialogue progresses, the dialogue manager then refines the high-level plan into low-level dialogue activities by considering the incrementally constructed dialogue context. The dialogue manager therefore interleaves high-level tutorial planning with on-the-fly situation-adaptive plan refinement and execution.

In addition to these tutoring specific requirements, the fact that the computer is participating in conversation with a human means it must perform the following dialogue management tasks as described by (Lewin, Rupp, Hieronymus, Milward, Larsson, & Berman, 2000) – *turn-taking management:* determining who can speak next, when, and for how long; *topic management*: determining what can be spoken about next; *utterance understanding*: understanding the content of an utterance in the context of previous dialogue; *intention understanding*: understanding the point or aim behind an utterance in the context of previous dialogue; *context maintenance*: maintaining a dialogue context; *intention generation*: generating a system objective given a current dialogue context; and *utterance generation*: generating a suitable form to express an intention in the current dialogue context. Although we focus on intention generation here, the other topics are discussed briefly as they are all inter-related, and are sometimes conflated in the literature. In the next section, we discuss the state of the art in dialogue management and why it is not sufficient to meet the requirements for managing tutorial dialogue.

2. STATE-OF-THE-ART DIALOGUE MANAGEMENT

We review three industrial-strength models of dialogue processing, namely, *finite state machines* (FSMs), *form-filling*, and *VoiceXML*, as well as an interesting cross-breed of FSMs and planning.

In the FSM approach, dialogue management is performed by a set of hierarchical FSMs that represent all possible dialogues. The top-level FSM is typically based on

the structure of the task to be performed (*e.g.,* flight information, phone banking transactions). For each state of the FSM, the dialogue engineer needs to specify its transitions to successor states, many of which handle numerous exceptions (*e.g.,* user help and cancel requests, timeouts). The dialogue engineer thus manually defines all possible dialogue flow; all alternative routes are drawn in full. Consequently, the construction of FSMs is not only domain specific but also labour intensive and prone to error. On the positive side, FSMs run in real time, and a well-designed FSM can produce help messages and issue re-prompts that are sensitive to the task context. On the negative side, the dialogues are system-driven: turn-taking as well as system feedback are hardwired, and there is only a limited and well-defined amount of dialogue context stored in each state of the network. This makes it hard to produce responses sensitive to unexpected input or to the linguistic context, and to provide personalised or customised advice or feedback. Also, it makes it hard to port a FSM-based dialogue system to a new domain. The size of a FSM is practically, not theoretically limited. A typical industrial dialogue system in the area of banking has circa 1500 states (personal communication, Arturo Trujillo, Vocalis plc). FSM construction is supported by various toolkits*, e.g.,* AT&T's FSM library (Mohri, Pereira, & Riley, http://www.research.att.com/sw/tools/fsm).

Form-filling is a less rigid approach to dialogue management. Instead of anticipating and encoding all plausible dialogues, the dialogue engineer specifies the information the dialogue system must obtain from the user as a set of *forms* composed of *slots*. The structural complexity of possible dialogues is limited only by the form design and the intelligence of the form interpretation and filling algorithm. This algorithm may be able to fill more than one slot at a time, maintain several active forms simultaneously, and switch among them. In contrast to a FSM-based dialogue system, the user of a form-filling dialogue system can therefore supply more information than the system requested (the system performs *question accommodati*on), or start a task before the system has offered to perform it (the system performs *task accommodation).*

VoiceXML (W3C, 2002) augments the form-filling approach with an XML-based specification language and support for speech input and output. It is an evolving industry standard that is designed for modeling audio dialogues including synthesised speech, digitised audio, spoken and DTMF key ("touch-tone") input, and mixed-initiative conversations. A VoiceXML document or set of documents defines a space of possible human-computer dialogues. Two types of dialogues are supported: *form*s that collect values for variables, and *menu*s that present the user with a choice of options. Each dialogue specifies the next dialogue in the sequence; if there is no successor, execution stops. *Subdialogue*s provide a mechanism for modularising common tasks such as confirmation sequences. Like a subroutine in a computer program, once the subdialogue is complete, the form interpreter returns to the place in the document where it was invoked. Subdialogues can be used to build libraries of short interactions shared among documents comprising a variety of applications. The acquisition and processing of *normal input* is complemented by an *event handler* that uses application-specific XML code to cope with *user help* and

cancel requests as well as with *no input* or *no match* situations. VoiceXML-based dialogues can be less rigid than our description suggests. The order in which the machine collects the information from the user is not entirely pre-determined. *Mixed-initiative dialogues* allow the user to provide information in a flexible order or to provide multiple pieces of information in succession without the interruption of intermediary prompts.

A major advantage of these three approaches is the robustness of their language understanding capabilities. Each approach has strong expectations about user input based on the state of the system. In the FSM approach, the nodes of the network can be associated with special grammars and language models; in the form-filling approach, one can associate actions with slot-filling events, for example, controlling the activation and combination of scoped grammars; in the VoiceXML approach, user input, provided in response to a system generated utterance produced by the interpretation of the contents of the <*prompt*> tag, is recognised using the grammar supplied by the associated <*grammar*> tag. The form elements <*help*> and <*catch*> are used to cover cases where the user fails to supply input or where the user input is not covered by the associated grammar.

All three approaches discussed so far face the problems of choosing among multiple refinements of a task, identifying which task the user is currently trying to perform, and detecting user-initiated switches among tasks or abandonment of tasks. These problems remain difficult and open. Moreover, FSM and form-filling approaches do not facilitate *per* se the maintenance of a dialogue history, which can then be exploited to support the generation of natural and effective feedback. The specification of VoiceXML, however, defines five distinct levels of variable scope (*session, application, document, dialogue,* and *anonymous*), allowing, in principle, the maintainance of a dialogue context. Used together with VoiceXML's conditional statements, they enable a dialogue engineer to implement a proper handling of, say, meta-dialogues and other linguistic phenomena (*e.g.*, generation of anaphora, ellipses).

In each of the three approaches, all plausible dialogues (FSMs) or their content (form filling, VoiceXML) have to be specified in advance. None of the approaches involve a deliberative component that can generate dialogue plans to achieve underlying goals, albeit (complex) forms can be seen as instantiated plans. A deliberative component also proves useful in cases where the execution of the first planned dialogue strategy fails and re-planning is needed.

Dialogue management in the AUTOROUTE system combines the power of deliberative planning with the benefits of FSMs in a two-tier approach (Lewin, 1998). The bottom-tier consists of *dialogue games*, which are *generic* FSMs. Dialogue games encode typical adjacency pairs (*e.g.*, a question is followed by an answer which may be followed by a confirmation and an acknowledgement) that do not specify the content of the dialogue moves they contain. In the top-tier, a deliberative agent treats the I/O behaviour of a dialogue game as a primitive action.

Given a goal, it generates a plan that has instantiated dialogue games as its primitive steps (*e.g.*, to pay a bill, obtain the date, amount, and bill-type; to obtain a date, play a question/answer game). Thus far, the AUTOROUTE approach has only been applied to the genre of information-seeking dialogues. Its planning architecture has not been fully exploited, and its re-planning capabilities allow only trivial cases of question and task accommodation. Moreover, turn taking is hardwired into the FSMs, which makes it hard to cope with situations where the user grabs the turn. Thus, the AUTOROUTE approach is a move in the right direction but more flexibility is required to handle tutorial dialogue.

3. A 3-LAYER ARCHITECTURE FOR MANAGING TUTORIAL DIALOGUE

Our three-layer architecture for managing tutorial dialogue has been inspired by studying planning architectures used in robotics.

3.1 Planning in Robotics

As we have seen, tutorial dialogue requires an architecture which can support the planning of feedback presentations that may consist of a sequence of dialogue actions to be delivered over several dialogue turns. During the multi-turn presentation, the system must monitor the success of the presentation, and respond effectively to events that indicate that extensions or modifications to the plan are necessary.

In the mid 1990's, a consensus architecture emerged in robotics. This architecture grew out of the realisation that reactive systems and hierarchical planning were not at odds with one another, but rather that in order to yield robust, flexible, and generalisable behavior, the strengths of these two methods needed to be combined. In purely reactive systems, robustness is gained at the expense of flexibility and adaptability. Action and perception are addressed, but cognition is ignored, limiting these robots to mimicking low-level life forms (Arkin, 1989). Extending the behaviors of such systems to more meaningful problem domains, requires memory and dynamic representations of the environment, as well as a representation of the goals the system is pursuing. This realisation led to the development of three-layer architectures that have become prevalent in robotics, and which we have adapted to the domain of dialogue.

(Bonasso, Firby, Gat, Kortenkamp, Miller, & Slack, 1997) describe a 3-layer planning architecture consisting of a *controller* (reactive component), a *sequencer* (hybrid component), and a *deliberator* (deliberative component). The controller is the component that interacts with the physical world. It possesses a library of primitive behaviors that it can execute. The controller's policy is to detect failure rather than to avoid it. The sequencer is a reactive plan execution and monitoring unit. In order to achieve tasks, it selects and sequences primitive behaviors (for execution by the controller) by taking the current situation or context into account. The sequencer has the capability to respond to contingencies (*e.g.,* a failure of a

primitive behaviour can be replaced by another primitive behaviour or a sequence thereof) and also to manage parallel interacting tasks. The deliberator is the planning component of the 3-layer architecture. It produces plans for the sequencer to refine and execute. It also looks for optimisations in the sequence maintained by the sequencer and can repair plans in response to exceptions.

To see this architecture in action, consider an office robot that is given the goal to take a worker's coffee mug to the kitchen sink. The robot's 3-layer planning architecture has the following division of labour. First, the deliberator decomposes the original goal and generates a high-level plan, say, (i) search the office for a coffee mug; (ii) find the door; (iii) exit safely; (iv) go to the kitchen; and (v) put the mug in the sink. This sequence of high-level steps is then passed to the sequencer, which performs a situation-dependent plan refinement. For example, the sequencer may refine step (iii), exit safely, to: (iii.a) move to the center of the office; (iii.b) point toward the door; (iii.c) follow the current heading with obstacle avoidance activated. Once the current step in the sequence is refined, its expansion is passed to the controller. The controller then executes primitive behaviours like obstacle avoidance, wall finding, wall alignment, following headings, and wandering.

Now reconsider Sect. 1.2, which contains our discussion of the many different tasks of managing dialogue in general, and tutorial dialogue, in particular. Given the proposed planning archicture, we need to identify the appropriate boundaries for the subdivision of functionality between the three planning layers. Also, we need to determine how we can achieve an effective coordination between the three layers. In order to address these questions and to validate and propagate our answers, we have built *BEETLE,* the ***B****asic* ***E****lectricity and* ***E****lectronics* ***T****utorial* ***L****earning* ***E****nvironment.* In the next section, we give an introduction to BEETLE's planning architecture. Sect. 4 then presents BEETLE's intelligent information presentation strategies from the perspective of its users.

3.2 A 3-layer Architecture for Managing Tutorial Dialogue.

Fig. 1 displays BEETLE's underlying generic and modular architecture for the management of tutorial dialogue. It is divided into four major parts (from left to right): external knowlege sources, the information state, the update engine, and the three-layered planning and execution architecture.

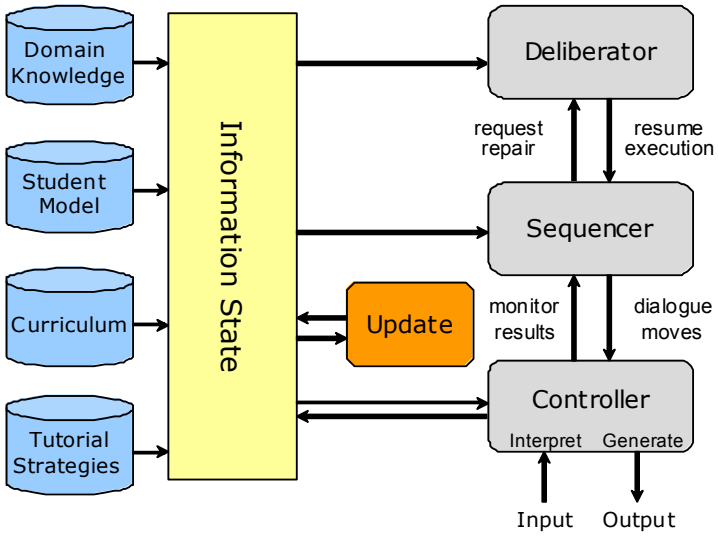

Figure 1. BEETLE's Dialogue Management Architecture.

BEETLE's architecture emphasises the importance of clearly separating the knowledge sources involved in tutorial dialogue. This aims to minimise the re-representation of knowledge in different parts of a tutor system. In addition, each knowledge source can be transparently accessed from other modules of the tutoring system via the information state (see below). The information state establishes an interlingua among modules with varying representations. The deliberator, for instance, can thus easily access domain knowledge, student modeling expertise, learning goals, and pedagogical strategies. Moreover, this design emphasises and encourages the reusability of components.

3.2.1 The Information State and the Update Module.

The *information state (IS)* captures the overall dialogue context and interfaces with external knowledge sources as well as with the planning modules. In particular, the IS contains a dialogue history that records all prior *dialogue moves* and the *common ground.* Dialogue moves refer to the conversational actions performed by an utterance. For example, the statement "I didn't understand you" requests information and signals non-understanding of the previous speaker's utterance. The common ground refers to the set of propositions that both dialogue participants have agreed upon in prior discourse. The IS also contains a list of salient objects to facilitate the treatment of linguistic phenomena such as anaphora. Moreover, the IS maintains a

stack of any pending discourse obligations, allowing the system to deduce the issues that it needs to, or intends to address in future dialogue continuations.

The IS is maintained by the *update module*. After each tutor and student turn, which usually consists of more than one dialogue move, this module is activated. Its *update rules* encode conversational expertise and define how to update the current context given a new dialogue move. Two example update rules are given below:

doDiagQuery:

precond:	latest turn contains move of type DIAG_QUERY
effects:	add DIAG_QUERY move to CDU add obligation for hearer to address move to CDU

doAssert:

precond:	latest turn contains move of type ASSERT asserted content addresses a previous DIAG_QUERY or INFO_REQ, say Q1
effects:	add ASSERT move to CDU remove speaker's obligation to address Q1 add hearer's obligation to address assertion add that speaker is committed to propositional content of assertion add that if hearer accepts assertion, then add assertion to common ground

The *update rule engine* fires the rule *doDiagQuery* if the latest turn contains a diagnostic query dialogue move. If so, that move would be entered into the current discourse unit (CDU), and an obligation for the hearer to address this move would be created. The rule *doAssert* fires if the latest turn contains an assertion and if its asserted content addresses a previous diagnostic query or information request. If this is the case, then the assertion is added to the CDU, the speaker's obligation to address the question is deleted, the hearer is now obliged to address the assertion, and the propositional content of the assertion is a candidate for entering the common ground.

3.2.2 Response Generation Using the 3-layer Architecture

The response generation module computes appropriate tutorial moves and synthesises tutorial feedback as text or other modalities. It uses the 3-layer planning architecture: a *deliberative planner* that projects the future and anticipates and solves problems (top layer); a *sequencer* or *plan execution and monitoring system* that performs adaptive on-the-fly refinement (middle layer); and a *controller* or *perception/action system* that interprets student actions and performs primitive tutor actions (bottom layer).

Top-layer: Deliberative Planning. The deliberation component synthesises plans from action descriptions, or *operators,* at the highest possible level of abstraction. This abstraction minimises wasted effort by allowing the middle layer to perform a maximum of situation-adaptive plan refinement. High-level planning results in a structured sequence of tasks that are passed to the *task agenda* of the middle layer; for each task, there is a *reactive action package* (see below) that achieves it, if successfully executed. The deliberator is explicitly activated in two cases: when the tutor begins a tutorial dialogue, and during the dialogue when the middle layer encounters a failure during plan execution and requests that the top layer performs a plan repair. However, the deliberative component also has a permanent background activity. It regularly inspects the agenda of the middle layer for two reasons: verifying whether pending discourse obligations (as recorded in the IS) are covered by the contents of the agenda, and anticipating problems or optimising the agenda's content. In both cases, the top layer can add or delete items from the task agenda as well as reorganise or aggregate them. The top and middle layer therefore need to synchronise their access to the agenda.

Fig. 2 depicts the operator *do_instruct_procedure* that can be used to instruct a student on how to perform a procedure. Its application leads to a high-level discourse plan with three main parts: first, it plans to open the discourse by communicating some initial thoughts about the procedure (*open_lab*); second, for each step in the domain plan, a discourse plan step *instruct_step* is generated; and third, after each step of the procedure has been instructed, *close_lab* evaluates the student's performance and terminates the lab with closing remarks.

```
schema do_instruct_procedure;
    vars          ?lab  = ?{type learning_goal},
                  ?args  = ?{satisfies listp},
                  ?domain_plan = ?{satisfies listp};
    expands       {instruct_procedure ?lab ?args};
    conditions    only_use_if {teaching_mode} = SOCRATIC,
                  compute {beer (has_planp ?lab ?args)} = t,
                  compute {beer (activate_plan ?lab ?args)} = success,
                  compute {beer (active_plan)} = ?domain_plan;
    nodes         1 action {open_lab ?lab},
                  2 iterate action {instruct_step ?step}
                       for ?step over ?domain_plan,
                  3 action {close_lab ?lab};
    orderings     1 ---> 2, 2 ---> 3;
    effects       {student_knows (lab ?lab)} = true at end_of self;
end_schema;
```

Figure 2. A High-Level Plan Operator for Instructing a Procedure (in OPLAN syntax)

The operator *do_instruct_procedure* is written in the task description language of OPLAN, the Open Planning Architecture (Currie & Tate, 1991). Consider its four

preconditions. The first condition, *{teaching_mode} = SOCRATIC,* ensures that it is only applied if the tutor is using a Socratic tutoring style. The other conditions interface with an external domain reasoner for Basic Electricity & Electronics, called BEER, which is queried three times. The first query checks whether BEER has a domain plan for the procedure the student is to perform; the second condition checks if BEER is able to activate the plan (NB: if a plan is activated, then BEER can be used to track the student's progress through its steps); and the third condition asks BEER to return a sequential representation of the domain plan. This plan is used to instantiate the *instruct_step* operator.

Fig. 3 displays a simplified version of the operator *instruct_step*. It consists of six steps: In the first four steps, it is planned that the tutor takes the turn, makes an assertion (namely that the next step is *?step)*, issues a directive telling the student to perform that step, and finally gives away the turn. It is then planned that the tutor waits for the student to react (*get_student_input*), and subsequently supplies feedback (*supply_feedback*) addressing the student's answer(s) or action(s).

```
schema do_instruct_step;
    vars          ?step      = ?{satisfies listp},
                  ?s_input = ?{satisfies listp};
    expands       {instruct_step ?step};
    nodes         1 action {take_turn},
                  2 action {assert (next_step ?step)},
                  3 action {direct (do ?step)},
                  4 action {give_away_turn},
                  5 action {get_student_input ?s_input},
                  6 action {supply_feedback ?s_input};
    orderings     1 ---> 2, 2 ---> 3, 3 ---> 4, 4 ---> 5, 5 ---> 6;
    effects       {student_knows (next_step ?step)} = true at end_of 2,
                  {student_performs (do ?step)} = true at end_of 3;
end_schema;
```

Figure 3. Instructing a Step

Currently, there is only one realisation of *instruct_procedure,* but there are several other realisations of *instruct_step*. An example of another decomposition asks the student whether he knows the next substep rather then simply asserting what the next substep is.

Middle-layer: Context-Driven Plan Refinement. *Reactive Action Packages*, introduced by (Firby, 1989), are the basic building blocks of a situation-driven plan refinement system. A reactive action package (RAP) groups together and describes all ways to carry out a specific task in different situations. In BEETLE, we are representing a RAP as a set of possible OPLAN action decompositions.

Our *sequencer*, the *OPLAN execution and monitoring environment,* executes the contents of the agenda, as filled by the deliberator. First, it selects the next task to be performed from the agenda. Then, it checks the selected task against the information state to see whether its effects have already been achieved. If this is the case, the task is deleted. Otherwise the sequencer identifies the RAP that can achieve the task. The methods of the identified RAP are checked, and the most appropriate of them is selected. If the chosen method results in a primitive action or a sequence thereof, then it is delegated to the bottom layer for execution; if the method is a complex task, then each of its subtasks is put on the agenda, and a new interpretation cycle starts.
The execution of a RAP can fail for three reasons: its preconditions are not met, none of its methods are applicable, or the execution of one of its primitive methods fails. The sequencer can cope with some failures, *e.g.*, it can try another applicable method. In the other cases, it has to call the top layer to cope with the failure.

Fig. 4 shows a RAP consisting of two OPLAN operators for the provision of tutorial feedback. The first operator is used to provide positive feedback to a correct student answer or GUI action; the second operator is applicable if the student's answer is incorrect. There are several other *supply_feedback* operators that cover cases where the student is partially correct, incorrect, or stuck and take other tutorial parameters (*e.g.,* student performance, tutor verbosity) into account as well.

```
schema do_supply_positive_feedback;
    vars            ?step  = ?{satisfies listp};
    expands         {supply_feedback ?step};
    conditions
      only_use_if {answer_correctness ?step} = CORRECT at begin_of self;
    nodes           1 action {play_positive_feedback_game ?step};
end_schema;

schema do_supply_feedback_question_certainty;
    vars            ?step  = ?{satisfies listp}, ?aut, ?app =?{satisfies number};
    expands         {supply_feedback ?step};
    conditions
      only_use_if  {answer_correctness ?step} = INCORRECT at begin_of self,
      compute {beesm (get_aut_app)} = {?aut, ?app},
      compute {?aut ~ 0.45} = t, compute {?app ~ 0.4} = t;
    nodes           1 action {take_turn},
                    2 action {diag_query (question_certainty ?step)},
                    3 action {give_away_turn},
                    4 action {get_student_input};
end_schema;
```

Figure 4. Two Decompositions for Supplying Feedback.

Bottom-layer: Perception/Action System. The bottom layer is responsible for the interpretation of student input and for the execution of primitive dialogue actions.
The *interpretation module* allows the student to interact with the system via text and graphical means. It identifies the meaning and intention behind a student utterance. This includes its syntactic analysis, the construction of a representation of its propositional content, the recognition of its speech act (*e.g.,* statement), the recognition of the intent behind the utterance (*i.e*., assertion and answer), and an evaluation of the student's utterance or action for correctness.
The *generation system* gets a sequence of elementary dialogue moves and micro-plans the generation of multi-modal feedback (natural language utterances and GUI actions). The bottom layer is supported by a sentence and media planner, which both have access to the full dialogue context. In particular, these components consult the list of salient objects and the contents of the previous and current discourse unit to generate natural feedback that makes use of elliptical constructions and anaphoric expressions. Action execution fails if the sentence or media planner fails.

3.2.3 Turn-Taking Management.

The tutoring agent releases the turn after it asks a question or requests that the student perform an action. In all other cases, the sequencer "cycles" until a question or action request is generated. If the student takes the initiative and grabs the turn, then this dialogue move will be recorded in the IS, generating an obligation for the tutor to address the last student utterance. The top-layer then deliberates over the new situation and may change the contents of the agenda accordingly. Similarly, if the student fails to react within a certain time limit, then the interpretation module generates an appropriate dialogue act, which the update engine processes, generating an obligation for the tutor to address the student's silence. The deliberative planner can then decide to either give the student more time, or to take the turn to supply help.

3.3 Implementation Status

We have built BEETLE, a prototype implementation of our computational framework for managing tutorial dialogue that serves to both validate and propagate our ideas. It is primarily based on two technologies, the TRINDIKIT dialogue system shell (Larsson & Traum, 2000) and the Open Agent Architecture (OAA) (Martin, Cheyer & Moran, 1999).
BEETLE's information state is entirely maintained and updated using TRINDIKIT. We were able to re-use the conversational expertise that was implemented in an information-seeking TRINDIKIT-based dialogue system, namely EDIS (Matheson, Poesio, & Traum, 2000). Many update rules were used unchanged; some were adapted for the purpose of tutoring, and new ones were written that capture dialogue moves that are only present in the tutorial dialogue genre (*e.g.,* hinting moves). TRINDIKIT also helps to glue together the different components of the system. Its language allows us to define the flow of control, and the information state serves as

a representation of the dialogue context that is shared among the various system components. Moreover, all major system components have been *agentified* with OAA wrapper code. OAA enables the dialogue system engineer to use components that are written in a variety of programming languages and run on different platforms. In line with our goal to provide a flexible, modular, and thus reusable architecture, domain reasoning is performed by a single agent. The BEER agent encodes BEETLE's knowledge about basic electricity and electronics. It has a rich LOOM (see http://www.isi.edu/isd/LOOM/LOOM-HOME.html) representation of BEE concepts, can perform basic inferences in this domain, and has explicit representations of domain plans. BEETLE's deliberative planner, OPLAN creates instantiated dialogue plans from its general dialogue strategies by accessing the relevant domain knowledge represented in BEER. As we described, OPLAN is used on both the top layer and the middle layer. Its deliberative capabilities have been extended by a plan execution and monitoring environment that implements a good part of the functionality of the sequencer: selecting the next element in the sequence, passing it down to the bottom layer, and monitoring its success. The sequencer can handle failures by asking the deliberative component to perform a plan repair. However, we need to extend the sequencer with a situation-adaptive plan refinement capability. At the time of writing, fully-fledged discourse plans are executed by the sequencer, frequently fail, and are therefore equally frequently repaired.

We have implemented an agent (BEESM) that determines the values for two variables that are used in constructing the system's response: *autonomy* and *approval*. Following (Brown & Levinson, 1987), autonomy captures the freedom of action and freedom of imposition by other agents; approval captures a positive consistent self-image that is appreciated and approved of by other agents. Applied to tutoring, for example, the tutor can ask the student whether he knows what to do next (high autonomy) or tell the student what to do next (low autonomy). Furthermore, a high approval response would be to praise a student answer whereas a low approval response would be to simply go on to the next topic. Autonomy and approval are determined by a Bayesian network that combines evidence from several situational factors derived from a study with teachers (*e.g.,* aptitude, correctness, material left, time left) (Porayska-Pomsta, Mellish, & Pain, 2000). BEEGLE, the user interface agent sends BEESM messages when relevant GUI actions occur (*e.g.,* button presses, lab actions) to allow BEESM to infer evidence of the presence of the situational factors.

BEETLE's perception and action system includes NUBEE, a natural language understanding module based on the CARMEL toolkit (Rose, 2000) and BEETLEGEN, an XSLT-based (W3C, 1999) sentence realiser.
The CARMEL toolkit comes with a spelling corrector, a robust parser, and a wide coverage English grammar and lexicon (COMLEX). The output of CARMEL is disambiguated and translated into BEETLE's logical form format, and references to objects are resolved. Reference resolution works by querying BEER for suitable objects matching the input description. In case of ambiguity, NUBEE can query the information state to determine which objects are most salient. Although CARMEL

has a wide coverage lexicon, it only contains syntactic and not semantic information. Thus, NUBEE's vocabulary is limited to the words for which we have provided semantics. As a start, we have covered common expressions for the elements in BEER and are now adding vocabulary such as "need", "think", and "want" not directly related to BEE concepts. In addition, we use Wordnet (Miller, 1990) to look for known synonyms of unknown words to help expand the coverage.
The natural language generation component, BEETLEGEN, performs pipelined XML-transformations for different stages of the generation process. The basic idea is to use XSLT stylesheets to transform a semantic input tree provided by OPLAN into a syntactic output tree that includes lexical items. The first step of the generation pipeline is to match the semantic input tree and build-up an initial syntactic structure, using a hybrid template-based and rule-based approach. The following stages perform NP-realization and make decisions about pronominalization as well as forming ellipses. The next step then performs lexical lookup including inflection for the non-canned parts of the generated tree structure. The last stage reads off the lexical items from the tree and produces polished English text (including proper punctuation and capitalization). BEETLEGEN's context-dependent pronominalization, its lexical variation, and its capability to omit redundant speech acts is supported by two agents, namely, the information state agent and the rudimentary student modelling agent BEESM.

4. THE BE&E TUTORIAL LEARNING ENVIRONMENT (BEETLE)

4.1 The Graphical User Interface

The starting point for BEETLE's graphical user interface was a course on basic electricity and electronics, which was written in HTML and the VIVIDS authoring environment (Munro, 1994) at the Navy Personnel Research and Development Center. This course included nine multiple-choice quizes and four labs requiring students to make basic measurements using a multimeter in a simulated environment. Two more complex labs involving building simple circuits and solving equations were added later (Rosé, Moore, VanLehn, & Allbritton, 2000). To integrate this curriculum with the rest of our system, it was necessary to reimplement the quizzes and labs and build an HTML viewer. This graphical interface, which also includes a chat window allowing typed dialogue with the system, is written in Tcl/Tk and uses add-on packages for the HTML display.

In interacting with BEETLE, students read textbook-style lessons written in HTML and then perform labs using the graphical user interface (GUI). In this section, we discuss in more detail the HTML pages, the quizes, and the labs.

BEETLE features a set of *BEE lessons*, each of which consists of hypertext pages that contain instructive text as well as static illustrations and animated images. Fig. 5

displays part of the hypertext lesson "characteristics of current". It features an animated image that illustrates the directed flow of current in a circuit. The illustration is supplemented by a paragraph of instructive text.

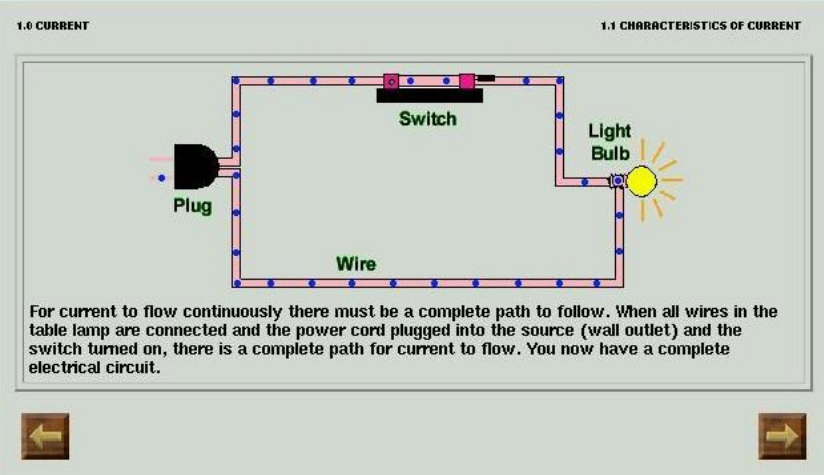

Figure 5. A BEE Hypertext Lesson Page on Measuring Current

The student can browse through the hypertext pages at his own pace, using the left and right arrows. Many pages such as the one in Fig. 5 are animated and some hypertext pages contain links that allow the student to explore the material in greater detail. The material is structured into lessons each of which has several sub-lessons ending in a *progress check*, a multiple-choice quiz designed to assess the student's understanding of the sub-lesson. Fig. 6 shows an example.

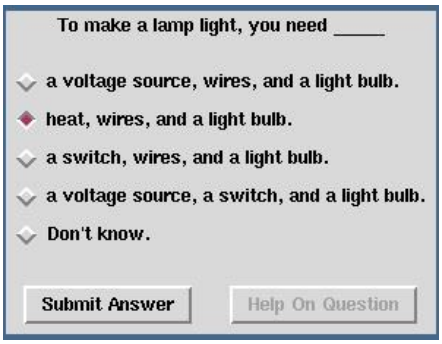

Figure 6. A Multiple-Choice Question.

Note, currently BEETLE has no tutorial strategies for helping students with the quizes but in future work these will be used to help students when they get a wrong

answer, click "Don't know" or "Help on Question". Currently, the student is simply told to try again if he gets a wrong answer and given the correct answer if he clicks "Don't know".

The current tutorial strategies focus on the labs. After the student successfully completes the theoretical part of a *lesson* (that is, reading a sequence of sub-lessons and answering the associated multiple-choice questions), he must complete a practical exercise, a so-called *lab*. Fig. 7 displays a screenshot of BEETLE while going through the lab *measure current* with a student. The screen is divided into four large areas: (top left) lessons done so far, (top right) the measuring current lab, (middle) the dialogue box, and (bottom) the student input entry window.

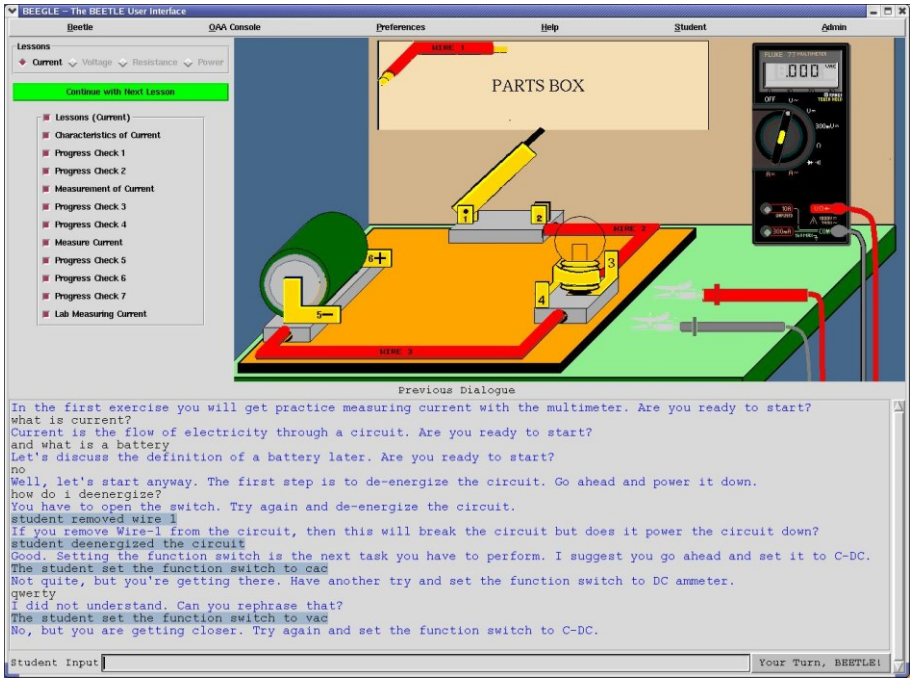

Figure 7. BEETLE Engaging in Tutorial Dialogue on Measuring Current.

The student can interact with the BEETLE interface in two ways. He can perform actions in the lab (*e.g.,* open the switch of the circuit, remove a wire, set the function switch of the multimeter) or type his questions and answers in English in the student input window. For example, the student can address a tutor question of the form "*do you know what to do next*" by typing his answer into the student input window, or by performing a GUI action that he thinks is the next step in the current procedure. BEETLE then supplies tutorial feedback in English by displaying it in the dialogue window.

4.2 Example Dialogue

Fig. 8 depicts an excerpt from a human-human tutorial dialogue, as it occurs in our BEE corpus. Note that this is a dialogue with a human tutor set within the VIVIDS learning environment.[1] The dialogue's topic is the multi-step procedure for measuring current. Earlier, the student read the relevant BEE lesson and answered a number of associated multiple-choice questions.

[1] **T**: Now connect the leads of the multimeter to the circuit!
[2] **S**: <student floundering>
[3a] **T**: OK,
[3b] can you tell me what the goal of this lab is?
[4] **S**: where do i connect the red lead?
[5] **T**: Before I answer that, answer the question I just asked you.
[6] **S**: to connect the multimeter in series
[7a] **T**: Good,
[7b] and what is the purpose of doing that?
[8] **S**: to measure current
[9] **T**: And do you remember what it means to hook something up in series?
[10a] **S**: yes,
[10b] i had to think for a minute.
[10c] let me try again
[11a] **T**: OK,
[11b] go ahead. ...

Figure 8. An Example Dialogue.

The excerpt is part of a larger dialogue context. Before this dialogue fragment began, the student interacted with the VIVIDS software. The student received a sequence of instructions and simple positive and negative multi-modal feedback. The student encountered no problems in successfully carrying out the first three instructions of the procedure *measure current* (de-energise the circuit, set the multimeter switch to CDC, and remove a wire), but then has problems connecting the leads of the multimeter to the appropriate places in the circuit. A human tutor then initiated the dialogue in Fig. 8. We now explain how the 3-layer architecture of the BEETLE system can play the role of the tutor in the dialogue.

[1] *(Rosé, Moore, VanLehn, & Allbritton,* 2000) *describe an experiment where students go through BEE lessons and labs with the guidance of a human tutor. The video signal from the student's computer was split so the tutor who was hidden behind a partition could watch the student's progress. The tutor and student were allowed to type messages to each other through a chat interface. We are using the logs of this chat interface, the* BEE dialogues*, to identify teaching tactics to be used by our tutor and plan to use them to train our system. The BE&E dialogues are publically available for download from our website (http://www.cogsci.ed.ac.uk/~jmoore/tutoring/dialogue-corpus.html) and also from the CIRCLE archives (http://www.pitt.edu/~circle/Archive.htm).*

When the tutoring session begins, the deliberative planner, OPLAN, is given the initial learning goal of teaching the student the procedure for measuring current. The planner, informed by its knowledge sources (*e.g.,* the BEER domain reasoner that provides a domain plan for measuring current) generates a high-level discourse plan:

T1: *open_lab*("measuring current")
T2: *instruct_step*("de-energise circuit")
T3: *instruct_step*("set multimeter switch to CDC")
T4: *instruct_step*("remove a wire")
T5: *instruct_step*("connect multimeter leads")
T6: *instruct_step*("energise the circuit")
T7: *instruct_step*("take meter reading")
T8: *close_lab*("measuring current")

While the high-level step *open_lab* can be fleshed out by the deliberative planner, the step *close_lab* has the effect of summarising the student's performance, and therefore cannot be planned in advance. In a similar vein, the deliberator does not need to expand any of the *instruct_step* tasks since the instruction of a step may depend on the student's prior performance. Once a high-level plan has been constructed, the deliberator then passes the sequence of tasks to the agenda of the the sequencer, which then starts processing the agenda's content.

Assume that the sequencer has successfully processed the tasks T1–T4, and that it is now choosing to process T5. The task *instruct_step* is complex, that is, not a primitive behaviour that can be executed by the controller. It must therefore be refined. Given the current dialogue situation, an applicable method for its decomposition is identified, and the sequencer produces the agenda:[2]

T51: *sequence* assert(next_step("connect multimeter lead"))
direct(do("connect multimeter lead"))
T52: supply_feedback(did("connect multimeter lead"))
T5: instruct_step("connect multimeter lead")
T6: instruct_step("energise the circuit")
T7: instruct_step("take meter reading")
T8: close_lab("measuring current")

After the sequencer starts another processing cycle, assume it chooses to process T51 only, which is an elementary task, and is therefore passed down to the bottom layer. The controller's feedback generator takes the assert and direct dialogue moves and generates a single English sentence to achieve their intended effects, in turn [1] of Fig. 8. Here, the assert move has been realised as an indirect effect of the directive. T51 is deleted from the agenda, and the update module adds the tutor's turn to the information state. Update rules, triggered by the turn, update the previous

[2] *The task T5 will be popped from the agenda only when the tasks T51 and T52 are finished.*

and current discourse units and add an obligation for the student to address the tutor's action directive.

The tutor releases the turn. Assume now that the student performs a number of GUI actions, none of which is connecting the multimeter leads to the circuit in a systematic and correct manner, and assume that the controller's *interpret* module posts the dialogue act "student floundering". The update module updates the information state by deleting the student's obligation to address the tutor's directive (the student did address the utterance, but was unable to perform it), and by adding an obligation for the tutor to address the student's floundering move.

If the task agenda contains a RAP designed to handle this event, then the sequencer will simply execute one of its applicable methods. Otherwise, the sequencer informs the top-level that it has no contingency for the situation and asks for a plan repair. Let us assume that the RAP *supply_feedback* can indeed handle the student's floundering, and that one of its applicable methods is to apply the *subgoal_reification_strategy*. The sequencer expands T52 into the tasks T521, T522, and T523, and we obtain:

T521: make_salient_goal_structure("measure current")
T522: make_salient(howto("connect multimeter leads"))
T523: direct("connect multimeter leads")

Task T521 is selected, it is decided that the goal structure is not yet or no longer salient, and T521 is expanded into:

T5211: diag_query("can you tell me what the goal of this lab is")
T5212: supply_feedback(answered("can you tell me what the goal of this lab is"))

Task T5211 is primitive, and therefore, it is passed to the bottom layer. The feedback generator now generates the utterances [3a] and [3b] of Fig. 7. Note that the sentence planner generates "OK" to resolve the tutor's pending low-level dialogue obligation to acknowledge the student's last move. It is now the student's turn, and he utters [4] instead of resolving his obligation to address the tutor's question. The update module therefore creates an obligation for the tutor to address this question, and obviously, T5212 is not applicable, nor are any other pending tasks in the agenda. The sequencer therefore calls the deliberator to cope with the situation.

The deliberator inspects the contents of the agenda as well as the IS and decides to resolve its obligation to address the student question by postponing its answer. Since the student did not address the tutor's question, the deliberator also decides to remind the student to answer its question [3b]. These two tasks are inserted into the task agenda, and since the sequencer identifies them as primitive, passes them to the feedback generator, yielding [5].

The controller's *interpret* module identifies the student's utterance in [6] as a partially correct answer to [3b]. The update module deletes the student's obligation to address [3b] and creates an obligation for the tutor to address the student's answer. The sequencer is able to address the new obligation by executing T5212. The RAP *supply_feedback* is expanded by a multi-turn tutoring strategy that is successfully carried out in [7a,7b,8].

The task T522 is to ask the student if he knows how to connect the leads; the sub-tasks T5221 (*diag_query*}) and T5222 (*supply_feedback*) are generated and executed to accomplish T522. Note that the sentence planner realises T5221 as [9] using the prior context, especially the student's utterance [6]. The student answers [9] with [10a], [10b], and [10c], the first two of which can be handled by T5222, producing [11a]. In [10c], the student asked the tutor for permission to try again, and the update module generated an obligation to address the student's request. Here, the planning engine falls back into deliberation since this obligation was not anticipated. The task T523 is replaced by the task T524, *grant_permission*("connect multimeter leads"). The bottom layer produces [11b] to accomplish T524, and the resulting agenda is then:

T52: supply_feedback(did("connect multimeter leads"))
T5: instruct_step("connect multimeter leads")
T6: instruct_step("energise the circuit")
T7: instruct_step("take meter reading"')
T8: close_lab("measuring current")

The tutor then waits for the student to connect the leads.

Note that the student could have reacted differently in each of his turns [2,4,6,8,10]. A flavour of BEETLE's dialogue flexibility and robustness can be gained from the interaction depicted in Fig. 7. Here, the student defied the system's expectations most of the time. BEETLE was able to handle each of the cases by a discourse plan repair in the planning engine or a low-level communication management action (a generation of a *request_rephrase* move) in the update module.

5. RELATED WORK IN TUTORIAL DIALOGUE SYSTEMS

In Sect. 2, we focused on previous work on dialogue management with sound theoretical underpinnings. This work was done in relatively simple domains such as information-seeking dialogues (*e.g.,* getting travel information) and action-seeking dialogues (*e.g*., phone banking transactions). The work on tutorial dialogue management is more *ad hoc;* we discuss it here to show that a more principled approach is necessary. We review the dialogue management in the tutoring systems AUTOTUTOR (domain: *computer literacy*) (Graesser, Wiemer-Hastings, Wiemer-Hastings, Kreuz, & Tutoring Research Group, University of Memphis, 1999), ATLAS-ANDES (*Newtonian mechanics*) (Schulze, Shelby, Treacy, Wintersgill, VanLehn, Gertner, 2000), and CIRCSIM/APE (*circulatory system*) (Khuwaja,

Evens, Michael, Rovick, 1994; Freedman, 2000) as well as in the EDGE explanation system (electrical devices) (Cawsey, 1989).

AUTOTUTOR's dialogue management can be regarded as an adaption of the form-filling approach to tutorial dialogue; to solve the feedback generation problem, it adds feedback moves to slots. AUTOTUTOR's dialogue management relies on a *curriculum script*, a sequence of *topic formats*, each of which contains a *main focal question*, and an *ideal complete answer*. The ideal complete answer consists of several sub-answers, called *aspects.* Each aspect includes: a good answer for that aspect, a list of anticipated bad answers corresponding to misconceptions and bugs along with corrections for those misconceptions and bugs; lists of prompts and hints that can be used to get the learner to contribute more information; and elaboration and summary moves that can be used to provide the learner with additional or summarising information. All of the moves are hard coded in English.
Using latent semantic analysis, AUTOTUTOR evaluates the student's answer to the main focal question against all the good answers of its ideal complete answer, and the anticipated bad answers. AUTOTUTOR gives immediate feedback based on the student's answer, and then executes dialogue moves that get the learner to contribute more information until all answer aspects are sufficiently covered. The category and content of tutor dialogue moves are computed by a set of 20 fuzzy production rules and an algorithm that selects the next answer aspect to focus on.
While AUTOTUTOR's dialogue management performs well in the descriptive domain of computer literacy, it is unclear how well this approach will work in problem-solving domains such as algebra or circuit trouble-shooting. In these domains student answers will often require the tutor to engage the student in a multi-turn remediation sub-dialogue. Curriculum scripts are not nested and do not allow the representation of multi-turn dialogues.

The dialogue manager of **ATLAS-ANDES** operates in the domain of Newtonian mechanics and thus must address AUTOTUTOR's aforementioned limitations. It uses a combination of *knowledge construction dialogues (KCDs),* which are recursive FSMs (Jordan, Rosé, & VanLehn, 2001) and a generative planner (Freedman, 2001). The grammar of a KCD definition bears many similarities to an AUTOTUTOR curriculum script. A student answer to a tutor question can be divided into correct and incorrect sub-answers with associated tutorial remediations. Unlike AUTOTUTOR, this feedback may extend over multiple turns through the use of recursive KCDs.
While AUTOTUTOR requires a pre-defined and hand-crafted curriculum script, the ATLAS-ANDES approach allows on-the-fly generation of nested KCDs, using the APE discourse planner. The ATLAS-ANDES architecture is therefore similar to the 2-tier AUTOROUTE architecture: its KCDs are dialogue games that are more complex but domain-specific. The simple generic question-answer pair is replaced by a recursive automaton that can deal with a specific question and many anticipated possible correct and incorrect sub-answers. A compiler then maps KCDs into plan operators. APE is then used to combine KCDs into larger recursive FSMs. The developers of ATLAS-ANDES propose a solution to the rigidity that is typically

associated with FSM-based systems. The reactive component of APE can skip around in the recursive KCD, for example, it can pop sub-networks that ATLAS-ANDES believes contain intentions that were already dealt with in prior dialogue.

In contrast, dialogue management in **EDGE** and **CIRCSIM/APE** is purely plan-based. EDGE provides two types of (STRIPS-like) operators: *discourse* and *content* operators. Discourse operators model Sinclair & Coulthard's four levels of discourse, namely, *transaction*, *exchange*, *move*, and *act* (Sinclair & Coulthard, 1975). Content operators specify how to construct explanations. For example, "to describe a device: explain its function, its structure, and its behaviour". EDGE's content operators are quite general; their bodies contain abstract domain references that interface with a knowledge representation module. EDGE incrementally builds and executes plans. Before each tutor turn, the deliberative planner expands the current unfinished step with either a complex sub-plan or an elementary plan step. Elementary plan steps are then executed using simple template driven generation. Thus, planning is delayed as much as possible so that the most current student model can be consulted.
CIRCSIM/APE also incrementally constructs and executes plans, and uses simple template driven generation for realising elementary plan steps. However, a major drawback of CIRCSIM/APE (for others, see (Freedman, 2001)) is that it embeds control in operators, unlike traditional planners, where control is separated from action descriptions. This makes writing operators difficult and puts an additional burden on the planner.

Although previous and ongoing work in tutorial dialogue systems has striven to support unconstrained natural language input and multi-turn tutorial strategies, there remain limitations that must be overcome: teaching strategies, encoded as curriculum scripts, KCDs, or plan operators, are domain-specific; the purely plan-based systems embed control in plan operators or, necessarily, conflate planning with student modeling and maintenance of the dialogue context; and all current tutorial dialogue systems except EDGE mix high-level tutorial planning with low-level communication management. These limitations can make systems difficult to maintain, extend, or reuse.

Looking at dialogue systems built by computational linguists and speech researchers, we see the opposite problem (Lewin, 1998; Allen, Byron, Dzikovska, Ferguson, Galescu, & Stenton, 2000; Pieraccini, Levin, & Eckert, 1997; Larsson, Ljungloef, Cooper, Engdahl, & Ericsson, 2000; Rudnicky & Wu, 1999; Chu-Carroll, 1999). These systems do not allow for conversational moves extending over multiple turns and the resulting need to abandon, suspend, or modify these moves. However, these systems aim for dialogue strategies that are independent of dialogue context management and communication management concerns. These strategies contain no domain knowledge; they query domain reasoners to fill in necessary details. Furthermore, in systems explicitly performing dialogue planning, control is never embedded in plan operators. In BEETLE, we have combined these beneficial features (modularity and re-usability) with the flexibility and robustness seen in tutorial systems.

6. CONCLUSION

In the introduction we argued that natural language dialogue is a critical modality for intelligent information presentation in the tutoring domain. This modality allows for multi-turn remediation subdialogues where students are encouraged to solve problems and detect and correct errors on their own. However, these subdialogues, while making this modality effective, also make it difficult to implement. Some current tutorial dialogue systems are able to support multi-turn remediation subdialogues; however, they do so by using carefully hand-crafted tutorial strategies that encode an initial question to ask and anticipated student responses and associated tutorial goals. The architecture for BEETLE splits the tasks necessary for supporting multi-turn remediation subdialogues among a set of independent modules that coordinate through a shared knowledge source (the information state). All domain knowledge is stored in the domain reasoner; all communication-management knowledge is encoded in the update module. We are currently populating a database of domain-independent tutorial strategies; we will test their effectiveness and portability against current domain-dependent approaches.

AFFILIATIONS

Claus Zinn
Johanna D. Moore
Mark G. Core

Human Communication Research Centre
School of Informatics
The University of Edinburgh
2 Buccleuch Place
Edinburgh EH8 9LW
United Kingdom

zinn@inf.ed.ac.uk
http://www.hcrc.ed.ac.uk/~zinn

J.Moore@ed.ac.uk
http://www.hcrc.ed.ac.uk/~jmoore

markc@inf.ed.ac.uk
http://www.hcrc.ed.ac.uk/~markc

REFERENCES

Allen, J., Byron, D., Dzikovska, M., Ferguson, G., Galescu, L., Stent, A. (2000). An architecture for a generic dialogue shell. *Natural Language Engineering* **6**.

Arkin, R.C. (1989). Towards the Unification of Navigational Planning and Reactive Control, Working Notes of the AAAI Spring Symposium on Robot Navigation, Stanford University, March 20-28.

Bloom, B.S. (1984). The 2 Sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. In *Educational Researcher,* Vol 13, pp. 4–16.

Bonasso, R.P., Firby, R.J., Gat, E., Kortenkamp, D., Miller, D.P., Slack, M.G. (1997). Experiences with an Architecture for Intelligent, Reactive Agents. *Journal of Experimental & Theoretical Artificial Intelligence.*

Brown, P., Levinson, S.C. (1987). Politeness: Some Universals in Language Usage. Cambridge University Press.

Cawsey, A. (1989). Explanatory dialogues. *Interacting with Computers* **1**, 69—92.

Chi, M.T.H, Bassok, M., Lewis, M.W., Reimann, P, Glaser, R. (1989). Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems. *Cognitive Science*, **13**(2), 145–182.

Chi, M.T.H., de Leeuw, N., Chiu, M., LaVancher, C. (1994). Eliciting Self-Explanations Improves Understanding. *Cognitive Science* **18**(3), 439–477.

Chi, M.T.H., Siler, S.A., Jeong, H., Yamauchi, T., Hausmann, R.G. (2001). Learning from human tutoring. Cognitive Science **25**, 471—533.

Chu-Carroll, J. (1999) Form-based reasoning for mixed-initiative dialogue management in information-query systems. In: Proceedings of the 7th European Conference on Speech Communication and Technology (Eurospeech-99), 1519 — 1522.

Cohen, P.A., Kulik, J.A., Kulik, C. (1982). Educational Outcomes of Tutoring: A meta-analysis of findings. *American Educational Research Journal* **19**, 237–248.

Currie, K., Tate, A. (1991). O-Plan: the open planning architecture. Artificial Intelligence **52**, 49—86.

Firby, R.J. (1989). Adaptive Execution in Complex Dynamic Domains. PhD thesis, Yale University, Technical Report YALEU/CSD/RR #672.

Fox, B. (1993). *The Human Tutorial Dialogue Project: Issues in the design of instructional systems*, Lawrence Erlbaum Associates, Hillsdale, New Jersey.

Freedman, R. (2000). Using a Reactive Planner as the Basis for a Dialogue Agent. *Proceedings of the Thirteenth Florida Artificial Intelligence Research Symposium (FLAIRS 2000), Orlando.*

Freedman, R. (2001). An approach to increasing programming efficiency in plan-based dialogue systems. In Moore, J.D., Redfield, C.L., Johnson, W.L., eds.: 10th International Conference on Artificial Intelligence in Education, IOS Press.

Graesser, A.C., Person, N. (1994). Question Asking During Tutoring. *American Educational Research Journal*. **31**(1), 104–137.

Graesser, A.C., Wiemer-Hastings, K., Wiemer-Hastings, P., Kreuz, R., Tutoring Research Group, University of Memphis (1999) AutoTutor: A simulation of a human tutor. *Cognitive Systems Research* **1**, 35—51.

Hume, G., Michael, J, Rovick, A, Evens, M. (1996) Hinting as a Tactic in One-on-One Tutoring, Journal of the Learning Sciences, **5**(1), 23–47.

Jordan, P.W., Rosé, C., VanLehn, K. (2001). Tools for authoring tutorial dialogue knowledge. In Moore, J.D., Redfield, C.L., Johnson, W.L., eds.: 10th International Conference on Artificial Intelligence in Education, IOS Press, 222—233.

Khuwaja, R.A., Evens, M.W., Michael, J.A., Rovick, A.A. (1994). Architecture of the CIRCSIM-tutor (v.3). In Proceedings of the 7th Annual IEEE Computer-Based Medical Systems Symposium, IEEE Computer Society Press, 158—163.

Larsson, S., Traum, D. (2000). Information state and dialogue management in the TRINDI dialogue move engine toolkit. Natural Language Engineering **6**, 323—340.

Larsson, S., Ljungloef, P., Cooper, R., Engdahl, E., Ericsson, S. (2000). GoDiS - an accommodating dialogue system. In: Proceedings of ANLP/NAACL-2000 Workshop on Conversational Systems.

Lepper, M.R., Chabay, R.W. (1988). Socializing the intelligent tutor: Bringing empathy to computer tutors. In *Learning Issues for Intelligent Tutoring Systems,* Mandl, H. & Lesgold, A. (editors). Springer, 114–137.

Lewin, I. (1998). Autoroute dialogue demonstrator. Technical Report CRC-073, SRI Cambridge.

Lewin, I., Rupp, C.J., Hieronymus, J., Milward, D., Larsson, S., & Berman, A. (2000). Siridus system architecure and interface report. Tech. report, Siridis D6.1.

Littman, D, Pinto, J, Soloway, E. (1990). The knowledge required for tutorial planning: An empirical analysis. *Interactive Learning Environments,* **1**, 124–151.

Matheson, C., Poesio, M, Traum, D. (2000). Modelling Grounding and Discourse Obligations Using Update Rules, In *Proceedings of NAACL 2000*, Seattle, 2000.

Martin, D.L., Cheyer, A.J., Moran, D.B. (1999). The open agent architecture: A framework for building distributed software systems. Applied Artificial Intelligence: An International Journal **13**, 91—128.

McArthur, D., Stasz, C., Zmuidzinas, M. (1990). Tutoring Techniques in Algebra. *Cognition and Instruction* **7**, 197–244.

Merrill, D.C., Reiser, B.J., Landes, S. (1992). Human tutoring: Pedagogical strategies and learning outcomes. Paper presented at the annual meeting of the American Educational Research Association, San Francisco.

Merrill, D.C., Reiser, B.J., Ranney, M., Trafton, G. (1992). Effective Tutoring Techniques: Comparison of human tutors and intelligent tutoring systems. *Journal of the Learning Sciences* **2**(3), 277–305.

Miller, G. (1990). WordNet: An on-line lexical database. *International Journal of Lexicography* **3**(4).

Munro, A. (1994). Authoring interactive graphical models, In: *The Use of Computer Models for Explication, Analysis and Experimental Learning*, Edited by deJong, T., Towne, D.M., Spada, H., Springer.

Ohlsson, S., Rees, E. (1991). The function of conceptual understanding in the learning of arithmetic procedure, *Cognition and Instruction,* **8**, 103–179.

Pieraccini, R., Levin, E., Eckert, W. (1997). AMICA: The AT&T mixed initiative conversational architecture. In: Proceedings of the 5th European Conference on Speech Communication and Technology (Eurospeech-97).

Porayska-Pomsta, K., Mellish, C., Pain, H. (2000). Pragmatic Analysis of Teachers' Language. Towards an Empirically Based Approach. In *Proceedings of AAAI Fall Symposium on Building Dialogue Systems for Tutorial Applications.*

Rosé, C. (2000). A framework for robust semantic interpretation, In Proc. of the 1st Annual Meeting of the North American Chapter of the ACL (NAACL-00), Seattle.

Rosé, C. P., Moore J. D., VanLehn, K., Allbritton, D. (2000), A Comparative Evaluation of Socratic versus Didactic Tutoring, University of Pittsburgh, LRDC-BEE-1.

Rudnicky, A., Xu, W. (1999). An agenda-based dialog management architecture for spoken language systems. In: Intl. Workshop on Automatic Speech Recognition and Understanding.

Schulze, K.G., Shelby, R.N., Treacy, D., Wintersgill, M.C., VanLehn, K., Gertner, A. (2000). Andes: A coached learning environment for classical newtonian physics. The Journal of Electronic Publishing **6**.

Sinclair, J.M., Coulthard, R.M. (1975). Towards an analysis of discourse: the English used by teachers and pupils. Oxford University Press.

VanLehn, K. (1990). Mind bugs: the origins of procedural misconception. MIT Press, Cambridge, Massachusetts.

W3C (1999). XSL Transformations (XSLT), Version 1.0. W3C Recommendation 16 November 1999, http://www.w3.org/TR/xslt

W3C (2002). Voice Extensible Markup Language (VoiceXML) Version 2. Working Draft, 24 April 2002, W3C Voice Browser Working Group. http://www.w3.org/TR/2002/WD-voicexml20-20020424/. Work in progress.

Young, R.M., Pollack, M.E., Moore, J.D. (1994). Decomposition and causality in partial order planning. In *Proceedings of the Second International Conference on Artificial Intelligence and Planning Systems*, pp. 188—193. Morgan Kaufman.

Zinn, C., Moore, J.D, Core, M.G. (2002), A 3-tier Planning Architecture for Managing Tutorial Dialogue, Intelligent Tutoring Systems, 6th Intl. Conference (ITS–2002), Springer LNCS 2363, 574—584, Biarritz, France.