

Model-Based Reasoning for Tutorial Dialogue in Shipboard Damage Control¹

Elizabeth Owen Bratt, Brady Clark, Zack Thomsen-Gray, Stanley Peters, Pucktada Treeratpituk, & Heather Pon-Barry

Center for the Study of Language Information
Stanford University
Stanford CA 94305-4115 USA
phone: 650-725-2319
{ebratt,bzack,peters,ztgray,pucktada, ponbarry}@csli.stanford.edu

Karl Schultz, David C. Wilkins, David Fried

Beckman Institute
University of Illinois
Urbana, Illinois 61801
{kaschult, dcw, fried}@uiuc.edu

ABSTRACT: The intelligent tutoring system based on the DC-TRAIN simulation requires specific information of various sorts about the student's performance during a session, in order to provide a focussed instructional review. The flexible dialogue architecture allows for dynamic system adaptation to student performance during the dialogue.

Keywords: simulation session action and knowledge representation for tutorial dialogue

1. Introduction

In this paper, we describe how our tutorial system extracts and uses information from the model of shipboard damage control provided by the DC-TRAIN simulation, and its summary of student actions compared to expert actions. Issues include designing appropriate representations and methods of accessing information from simulator sessions. We discuss our system architecture, and how our flexible dialogue management and intelligent tutor module provide effective instructional dialogue that adapts to each student's DC-TRAIN session and performance within the dialogue.

2. DC-TRAIN for Shipboard Damage Control

The DC-TRAIN system (Bulitko & Wilkins 1999) supports qualitative reasoning in training damage control assistants (DCA's) to manage shipboard crises appropriately, using general principles of how and where fires might spread, likely consequences of fires (smoke) and firefighting (floods), and tradeoffs in resource use involving strategically important compartments such as magazine rooms filled with explosive weapons and large vs. small compartments.

¹ This work is supported by the Department of the Navy under research grant N000140010660, a multi-disciplinary university research initiative on natural language interaction with intelligent tutoring systems

The DC-TRAIN system involves a simulation of a DDG-51 Arleigh Burke destroyer, its firemain, and the properties of its compartments with respect to fires. Student DCA's can use DC-TRAIN for running through scenarios to test their ability to handle the fire, flood, smoke, and firemain crises produced by various scenarios, in reasonable speed and with the most effective containment of the crises. A separate DCX (damage control expert) agent component can run at the same time as the DC-TRAIN simulation to provide the actions of an idealized DCA, based on rules for optimal response to each crisis. The DCX agent generates a comparison of the expert actions with the student's actions, in the form of a database table called the Expert Session Summary Graph (ESSG).

3. Tutorial System

As a further training component, we are developing a tutorial system to conduct after action review with the student DCA, based on the ESSG record of each of their sessions with DC-TRAIN. We discuss first some of the issues in extracting and representing knowledge about the student's DC-TRAIN session, then outline our overall system architecture, with more detailed discussion of our dialogue management and tutoring module's structure and strategies.

3.1 Knowledge Representation Issues

Ensuring that the ESSG contains adequate information for the tutorial component to review with the student has been an important design goal of recent work. Some of the necessary modifications to the ESSG have been:

- providing an explicit representation of the causal link between original crises in a scenario, and resulting crises caused by fires spreading, fires producing smoke, fire fighting efforts producing flooding, etc. This allows the tutorial system to present its discussion of related crises in a coherent fashion, and to have a measure of how severe the results were from a crisis from the scenario.
- for each action or report in the system, providing a representation of the main action or statement, plus values for each of the possible parameters. This allows the tutorial system to speak about comparisons of the student's actions with the correct action, as well as to have the means to aggregate related actions, either of the DCX or the student, for purposes of discussion. The aggregation may be necessary either for the system's plan of how to conduct the review, or for the system to respond to the student's questions. The discussion may be of what crises had similar mistakes, similar recommended actions, or similar use of resources such as the same repair teams being involved.
- providing a representation of the goals which a particular action may serve, from immediate and direct goals, to higher level ones.
- tolerance to unimportant differences from DCX, such as when the student chooses to start a different fire pump from the one that the DCX chose, but the student's pump choice meets all applicable constraints.

DC-TRAIN produces the initial representation of the ESSG information in a Microsoft Access database table, which we translate into a Java class for easy querying by our Tutoring Module (discussed in Section 3.4).

Beyond these issues of representing the ESSG information appropriately, we also are addressing issues of visualization of information for presentation during the tutorial session. A ShipDisplay lights up compartments with various colors indicating the type of crisis during DC-TRAIN. The tutorial system aims to use the same display to highlight items during discussion. This could be especially valuable for discussing items of spatial reasoning, such as the proper placement of fire boundaries and tracing the spread of a fire or flood and discussing considerations in managing that crisis at various stages of its development.

3.2 System Architecture

In this section, we discuss some models and techniques we used to deal with the complex structure of fire, flood, smoke, and firemain crises produced by various DC-TRAIN scenarios.

To facilitate the implementation of multi-modal, mixed-initiative interactions, we implemented our system within the Open Agent Architecture (OAA) (Martin et al. 1999). OAA is a framework for coordinating multiple asynchronous communicating processes. The core of OAA is a 'facilitator' which manages message passing between a number of encapsulated software agents that specialize in certain tasks (e.g., speech recognition).

Our system uses OAA to coordinate agents for the following components of the system:

- **Gemini** natural language understanding (Dowding et al. 1993). Gemini uses a single unification grammar both for *parsing* strings of words into logical forms (LFs) and for *generating* sentences from LF inputs. This agent enables us to give precise and reliable meaning representations which allow us to identify the discourse move types (e.g., a question) given a linguistic input or output; e.g., the question "What happened next?" has the LF: (ask(wh([past,happen]))).
- **Nuance** speech recognition, which converts spoken utterances to strings of words. Nuance relies on a language model, which is compiled directly from the Gemini grammar, ensuring that every recognized utterance is assigned an LF.
- **Festival** speech synthesis, which 'speaks' word strings generated by Gemini.
- A Dialogue Manager, which coordinates inputs from the user, interprets the user's dialogue moves, updates the dialogue context, and delivers speech and graphical outputs to the user. This agent is discussed in detail in Section 3.3.
- A **Tutoring Module**, which identifies appropriate tutorial strategies (positive or negative feedback, follow-up questions, hints) in response to user input. This agent is discussed in detail in Section 3.4.

Agents 1-3 are 'off-the-shelf' dialogue system components (apart from the Gemini grammar, which must be modified for each application). We implemented agents 4 and 5 in Java specifically for this application. This OAA/Gemini/Nuance/Festival architecture has been deployed successfully in an unmanned helicopter interface (Lemon et al. 2001).

3.3 Dialogue Manager

3.3.1 Dialogue Management Architecture

The Dialogue Manager makes use of several recent ideas in dialogue modeling, described in detail in Lemon et al. (2001). It creates and updates an *Information State*, corresponding to a notion of dialogue context. Dialogue moves (e.g., an assertion) update the information state and are mixed-initiative. A dialogue move might send a response to the tutoring system, elicit feedback from the tutoring system, or prompt a follow-up question.

Figure 1 provides a sample of the type of tutorial interaction our system aims to support.

T=tutor, S=student	
T	There were two fires, and you put out both of them. You also handled one flood. The first fire was in the officer's baggage room. For a fire, what's the first thing you should do?
S	Send a repair team to set fire boundaries on the compartment.
T	No, you missed a step. What comes before setting fire boundaries?
S	I'm not sure.
T	Okay. First send a team to investigate. For the second fire, you also forgot to send a team to investigate. Okay. After setting fire boundaries, what should you do?
S	Order a repair team to fight the fire.
T	No, you missed a step. What should you do before sending a repair team to fight the fire?

S	Isolate the space.
T	Yes. Always electrically and mechanically isolate a space before fighting a fire. For the second fire, you also forgot to isolate the space.

Figure 1: Sample Dialogue

The Dialogue Manager includes two types of declarative knowledge: dialogue move types and the activity model. Dialogue move types are Java classes and include *assertion*, *wh-query*, *yes-no-query*, *yes-no-response*, etc.

The Activity Model is a hierarchical and temporal decomposition of tutorial activity. Unlike other aspects of the dialogue management architecture, only the Activity Model is specific to tutorial activity. The Activity Model includes *activity types*. Activity types are Java classes and include EXPAND_ONLY, SUMMARY (i.e., a statement that does not require a response), YN_QUERY (e.g., a question like “Ready to begin reviewing your session?”), and WH_QUERY (e.g., “What comes before setting fire boundaries?”). EXPAND_ONLY activities are more like placeholders for scripted sets of actions the tutor plans to perform. For example, the EXPAND_ONLY activity type ‘start’ corresponds to a sequence of actions: a YN_QUERY ‘ready to begin’, an EXPAND_ONLY node ‘review session’, and a SUMMARY (‘goodbye’). Aside from EXPAND_ONLY, activity types represent primitive actions like asking a yes-no question.

The Dialogue Manager includes the following dynamically updated components:

- A *Dialogue Move Tree*: a structured history of dialogue moves and ‘threads’, plus a list of ‘active nodes’.
- An *Activity Tree*: a temporal and hierarchical structure of activities initiated by the system or the user, plus their execution status.
- A *System Agenda*: the issues to be raised by the system
- *Saliency Groups*: the objects referenced in the dialogue thus far, ordered by recency (Fry et al. 1998)
- *Pending List*: the questions asked but not yet answered
- *Modality Buffer*: stores gestures for later resolution

In the Dialogue Move Tree, each node is of a particular dialogue move type. Each node in the tree is a dialogue move, either by the tutor or by the student. Nodes in a dominance relation correspond to sub-dialogues; e.g., when the tutor asks a follow-up question in response to the student’s answer to a question like *What should you do in response to a fire alarm?*

In the Activity Tree, each node is of a particular activity type. Nodes are constructed and added to the tree by the Tutoring Module (see Section 3.4 below). At the initial part of the session, the system creates a single EXPAND_ONLY node called ‘start’ and adds it to the tree. It then begins executing tasks. The activity ‘start’ is passed to a hashtable which returns the corresponding sequence of actions: a YN_QUERY ‘ready to begin’, an EXPAND_ONLY node ‘review session’, and a SUMMARY ‘goodbye’. These nodes are all added to the tree and the YN_QUERY is asked. The tutor then waits for a response from the student. Based on that response, the tutor adds a new action to the tree and executes again or simply executes the next task.

3.3.2 Benefits

There are several benefits to our dialogue management architecture:

- The dialogue management architecture is reusable across domains. As mentioned, the same architecture has been successfully implemented in an unmanned helicopter interface (Lemon et al. 2001). The activity model--- e.g., the properties of the relevant activities--- will have to be changed across domains.
- The Dialogue Tree/Activity Tree distinction allows one to capture the notion that dialogue works in service of the activity the participants are engaged in. That is, the structure of the dialogue, as reflected in the Dialogue Move Tree, is a by-product of other aspects of the dialogue management architecture; e.g., the Activity Model and the Tutoring Module. The Dialogue Tree/Activity Tree distinction is supported by recent theories of dialogue; e.g., Clark’s (1996) joint activity theory of dialogue.

- The dialogue move types are domain-general, and thus reusable in other domains.
- The architecture supports multi-modality with the Modality Buffer. For example, we are able, in principle, to coordinate linguistic input and output (e.g., speech) with non-linguistic input and output (e.g., the user can indicate a point on a map with a mouse click or the system can illuminate a point on a map).

3.4 The Tutoring Module

3.4.1 Tutoring Module Components

The Tutoring Module processes the ESSG in order to discuss the student's DC-TRAIN session based upon the structure, causality, and behavior of its components (i.e., crises and actions). This information is used both to construct an overall tutoring strategy and to determine appropriate reactions to student input.

Before any tutor-student interaction takes place, the Tutoring Module devises the aforementioned tutoring strategy. This strategy composes the EXPAND_ONLY node 'review_session' which is added to the Activity Tree as a part of the 'start' node. This node defines which crises (out of the total set in the session) the tutor will discuss with the student. Crises are eliminated if the student performed perfectly in response to that crisis, or if a crisis with similar errors has been chosen for discussion. A given scenario may have many of the same type of crisis, so this preprocessing of the ESSG into a tutoring strategy seeks to avoid redundancy and keep the student interested.

The Tutoring Module must thus perform an analysis of the ESSG while constructing this strategy. Similar crisis types must be identified, as well as similar student errors. It is important to note that repeat errors are not only relevant in the strategy construction; they are also important to mention to the student during the tutoring dialogue itself (as seen in Figure 1). We hope to identify “exemplar” crises in the session. These are those crises that represent the poorest student performance on that given crisis type. These crises will make for the most interesting dialogues and provide the student with the most opportunity for learning. It may be the case that a student performs perfectly on all of a given crisis type save one and we may need to take this into account when evaluating the students performance.

As noted above, the Tutoring Module identifies appropriate strategies in response to user input. The Tutoring Module includes one type of declarative knowledge: a library of tutoring strategies. Figure 2 illustrates a tutoring strategy. For legibility, the key elements are presented in English rather than Java. Figure 3 formats the strategy in Figure 2 as a dialogue.

```
def_strategy discuss_error_of_omission_answer_incorrect
: goal (did_discuss_error_of_omission_answer_incorrect)
: preconditions
  (i) the student's answer is incorrect
  (ii) the student's actions in response to the damage event included an error of omission
: recipe
  (i) provide negative feedback to the student
  (ii) give the student a hint
  (iii) ask a follow-up question
  (iv) classify the student's response
  (v) provide feedback to the student
  (vi) tell the student the rule
  (vii) tell the student that the topic is changing
```

Figure 2: Sample tutorial strategy

T	The first fire was in the chemical warfare defense equipment storeroom No. 2. What is the first thing you should do in response to this crisis?
---	---

S	Send a team to isolate the compartment.
T	No, that incorrect. You missed a step. What should you do before isolating the compartment?
S	I don't know.
T	You should investigate the compartment. Let's move on.

Figure 3: Sample tutorial strategy dialogue

To initiate a tutoring strategy, the student invokes the Tutoring Module by responding to a question from the tutor; e.g., “What should you do in response to a fire alarm?” The system searches the library of a tutoring strategies to find all strategies whose preconditions are satisfied in the current context. Like the plan operators in other systems (e.g., Atlas/Andes; Freedman 2000), each tutorial strategy has a multi-step *recipe* (Wilkins 1988) composed of a sequence of actions. Actions in a recipe can be primitive actions like providing feedback or complex actions like embedded tutorial strategies.

The Tutoring Module utilizes information in the ESSG to decide which tutorial strategy is appropriate with respect to a student's response to a particular question. For example, in Figure 2, the Tutoring Module uses, in the preconditions on the application of the tutoring strategy, the information in the ESSG which classifies the relevant action as an error of omission, in addition to the classification of the student's response as an incorrect answer. The preconditions on other tutoring strategies will involve different combinations of action and response classification. Hence, it is the combination of the classification of a student's response (as correct, incorrect, etc.) and action in a DC-TRAIN session (as an error of omission, error of commission, etc.) which determine which tutoring strategy the Tutoring Module uses to teach the student.

The response classification mentioned above is not trivial. The student response to the tutor's query and the ideal student response as stored in the ESSG are in dissimilar forms. They are both translated into Command objects which represent the actions in question. The two objects can then be easily compared. Comparisons are made based on each relevant field of the object - the order type (e.g., setting flood boundaries), the agent (e.g., repair team 5), and the location (e.g., the laundry room). Student responses that do not match the ideal response in one field will be dealt with differently than student responses that do not match the ideal response in other fields. For example, a response with the wrong order type is a more serious error than a response with only the wrong agent.

The Tutoring Module makes uses of other aspects of the ESSG when tutoring the student; e.g., the causal structure of the model. Some actions in DC-TRAIN are steps in sequences of actions. For example, when fighting a fire the DCA must send a repair team to investigate the compartment, isolate the compartment, set fire boundaries, and fight the fire, in that order. If the student incorrectly omits one of these steps, while still succeeding in completing the other actions in the sequence, this is an error of omission. Students may also fail to identify the correct step in a causal sequence when responding to a tutor's question. The Tutoring Module uses this type of causal information contained in the ESSG to determine the appropriate type of hint (e.g., “You missed a step”) to give to a student.

4.0 Conclusion

Tutoring sessions use a complex representation of student DC-TRAIN performance, and structure it to provide an effective, focused review. The flexible dialogue architecture allows for the system to adapt to student performance during the dialogue, taking patterns of action from the DC-TRAIN session into account, both in formulating dialogue responses and in creating dialogue structure.

References

- Bulitko, V.V. and D.C. Wilkins. 1999. Automated instructor assistant for ship damage control. *Proceedings of AAAI-99*
- Clark, H.H. 1996. *Using Language*. Cambridge University Press.

Dowding, J., J. Gawron, D. Appelt, J. Bear, L. Cherny, R.C. Moore and D. Moran. 1993. Gemini: A natural language system for spoken-language understanding. *Proceedings of the ARPA Workshop on Human Language Technology*.

Freedman, Reva. 2000. Plan-Based Dialogue Management in a Physics Tutor. *Proceedings of the Sixth Applied Natural Language Processing Conference (ANLP '00)*.

Fry, J., H. Asoh and T. Matsui. 1998. Natural Dialogue with the Jijo-2 Office Robot. *Proceedings of IROS-98*: 1278-1283.

Lemon, O., A. Bracy, A. Gruenstein and S. Peters. 2001. *Proceedings Bi-Dialog, 5th Workshop on Formal Semantics and Pragmatics of Dialogue*:57-67.

Martin, D., A. Cheyer and D. Moran. 1999. The Open Agent Architecture: a framework for building distributed software systems. *Applied Artificial Intelligence* 13, 1-2.

Wilkins, D. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm* . San Mateo, CA: Morgan Kaufmann.

