Using a Reactive Planner as the Basis for a Dialogue Agent

Reva Freedman

LRDC #819
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15260

freedrk+@pitt.edu
http://www.pitt.edu/~freedrk

Abstract

This paper describes APE (the Atlas Planning Engine), the reactive planner at the center of the Atlas dialogue management system. The goal of Atlas is to build conversation-based systems, where turns in the "conversation" may include graphical actions and/or text. Since APE can be used to generate a dialogue involving arbitrarily nested discourse constructs, it is more powerful than dialogue planners based on finite-state machines. Although it is intended largely to model dialogue containing hierarchical, multi-turn plans, APE can also be used as a general-purpose programming tool for implementing a dialogue system.

Introduction

Dialogues such as the one in Figure 1 play an important role in text-based applications involving communication between a person and a computer, such as intelligent tutoring systems (ITSs), advice-giving systems and interactive help systems. In the past, such systems have often been implemented with finite-state machines, either simple or augmented. But finite-state machines do not permit one to model general discourse structures, since the latter can contain arbitrarily nested components.

Yet neither an algorithm based on a context-free grammar nor a STRIPS-style planner is a good choice for modeling a conversation either. Primarily, one cannot fully plan a conversation in advance because it is impossible to predict what the other agent is going to say. Even if it were possible to enumerate all the alternatives in advance, e.g. using a menu-based input system, it is inefficient to elaborate sections of a plan that will never be executed.

For these reasons we have chosen reactive planning (Georgeff and Ingrand 1989) as the underlying model for our dialogue planner. In short, we model conversation like a chess game: expert players have a plan for upcoming moves, but they are also capable of changing the plan when circumstances change.

Our plan operators are based largely on the hierarchical task network (HTN) style of planning (Yang 1990; Erol, Hendler and Nau 1994). First, decomposition is better suited to the type of large-scale dialogue planning we are currently doing than means-end reasoning. It is much easier to establish what a human speaker will say in a given situation than to be able to understand why in sufficient detail and generality to do means-end planning. Second, the conversation is in a certain sense the trace of the plan. In other words, we care much more about the actions generated by the planner than the states involved, whether implicitly or explicitly specified. Finally, we view dialogues as having a hierarchical structure (Grosz 1977), and we use the hierarchy information in preconditions as one way to maintain coherence.

In this paper we will describe APE (the Atlas Planning Engine), a reactive dialogue planner we have built for Atlas, a conversation manager for intelligent tutoring systems (Freedman 1998). APE is a descendant of the system described in Freedman and Evens (1996). We will provide an annotated example of dialogue generation to show how APE can be used to generate dialogues that earlier systems could not handle.

Operation of the Planner

Figure 2 contains examples of plan operators. The system stores its intentions in an *agenda*, which is implemented as a stack with additional operations available besides *push* and *pop*. In addition to a goal, a stack entry can include a selected plan operator and its bindings. To initiate a planning session, the user invokes the planner with a goal, e.g. to conduct a conversation covering a specified topic, or to help a student solve a problem. The system stores the initial goal on the agenda, then searches the operator library to find all operators whose *goal* field matches the goal on top of the agenda and whose *filter conditions* and

This research was supported by NSF grant number 9720359 to CIRCLE, the Center for Interdisciplinary Research on Constructive Learning Environments at the University of Pittsburgh and Carnegie-Mellon University.

preconditions are satisfied. The filter slot is used for static properties, such as properties of domain objects, while preconditions are used for characteristics that can change as the dialogue progresses, e.g. the number of times a particular construct has been used. (For simplicity, the filter slot is not shown in the figures.)

Goals are represented using first-order logic without quantifiers, with unification used for matching. Multiple conditions can be used, and are considered to be *and*-ed together. The *not* function is also available; the closed world assumption is used. If more than one ⟨operator, binding list⟩ match is found, the last one found is used, although the user has the option of writing a different algorithm for resolving duplicates.

Since the system is intended especially for generation of hierarchically organized task-oriented discourse, each operator has a multi-step *recipe* in the style of Wilkins (1988). When a match is found, the matching goal is removed from the agenda and is replaced by the following items (top of list represents top of agenda):

- ⟨BEGIN⟩ marker
- First step of recipe
- Second step of recipe
- . . .
- ⟨END⟩ marker

The ⟨END⟩ marker contains the goal that triggered the operator in case we later need to find another way to satisfy the same goal.

Recipe items can take several forms:

- *Goal:* Create a subgoal.
- *Primitive:* Do an action. Since this is a text planner, the action is usually to say something. Graphical actions can be implemented as primitives or via the use of an external relation, described later.
- *Interactive primitive:* Say something and give control to the other party to reply.
- *Assert:* Add a ground fact to the transient knowledge base.
- *Retract:* Remove all matching facts from the transient knowledge base.
- *Fact:* Evaluate a condition. If false, skip the rest of the recipe.
- *Retry-at:* Pop the agenda through the first ⟨END⟩ marker where the retry argument is false, then restore that entry's original goal.
- *Prune-replace:* Pop the agenda until the retry argument becomes false, then push an optional list of new recipe items onto the agenda.

Fact is used to allow run-time decision making by bypassing the rest of an operator when circumstances change during its execution. Although preconditions and *fact* both allow one to express *if-then* conditions in the APE syntax, preconditions are only checked when an operator is being chosen. Although the use of *fact* could be replaced by preconditions on an additional operator, the use of *fact* allows a more natural writing style.

Retry-at implements a Prolog-like choice of alternatives. If there are multiple ways to satisfy a goal, *retry-at* allows one to choose among them a second time if one's first choice is later shown to be undesirable. For *retry-at* to be useful, the author must provide multiple operators for the same goal. Each operator must have a set of preconditions enabling it to be chosen at the appropriate time. As in Prolog, *retry-at* is also useful for implementing loops; one simply chooses the same operator every time until the exit condition is reached. *Fact* can be used to check the exit condition.

Prune-replace allows a type of decision-making frequently used in dialogue generation. When a conversation partner does not give the expected response, one would often like to pop the top goal from the agenda and replace it with one or more replacement goals. *Prune-replace* implements a generalized version of this concept, allowing one to pop the agenda until a desired configuration is reached. Although any instance of *prune-replace* could be replaced by *retry-at* and an additional operator, the use of *prune-replace* permits a more natural writing style.

Data Storage

The APE environment contains two ground-clause knowledge bases. One contains permanent information such as domain facts, while the other contains transient facts that become true during program execution. Although APE puts no restriction on relation names, the following naming convention is used in the examples below to clarify

1 T: What controls TPR?
2 S: Arteriolar radius. *(near miss)*
3 T: And what controls arteriolar radius?
4 S: Radius of arterioles. *(near miss)*
5 T: No, radius of arterioles and arteriolar radius are the same thing.
So what controls arteriolar radius?
6 S: Arteriolar diameter. *(near miss)*
7 T: Arteriolar diameter and arteriolar radius are the same thing in different units.
Try again. What controls arteriolar radius?
8 S: Arteriolar resistance. *(near miss)*
9 T: No, actually arteriolar radius controls arteriolar resistance.
So what controls arteriolar radius?
10 S: Sympathetic stimulation.
11 T: In other words, the nervous system.

Figure 1: Sample dialogue

Operator T1:

```
(def-operator neural-partial-mechanism
  :goal (did-handle-near-miss)
  :precond ((w-variable-is ?neural)
            (is-neural ?neural)
            (w-topic-is mechanism)
            (w-level-is topic)
            (i-input-concept-is ?partial))
  :recipe  ((goal (did-near-miss-specific ?neural ?partial)))
  :hiercx  ())
```

Operator T2:

```
(def-operator neural-partial-mechanism-first
  :goal (did-near-miss-specific ?neural ?partial)
  :precond ((not (w-max-nearmiss-is ?max))
            (e-on-leg ?partial ?neural none))
  :recipe  ((assert (w-max-nearmiss-is ?partial))
            (goal (did-tutor-deep mechanism ?partial ?neural t)))
  :hiercx  ())
```

Operator T3:

```
(def-operator neural-partial-mechanism-later-synonym-execution
  :goal (did-near-miss-synonym ?neural ?partial ?max)
  :precond ()
  :recipe  ((goal (did-utter ("No, " ?partial "and" ?max "are the same.")))
            (prune-replace ((w-level-is prompt)
              (goal (did-tutor-deep mechanism ?max ?neural post-error)))))
  :hiercx  ())
```

Operator T6:

```
(def-operator mech-ling-near-miss
  :goal (did-handle-ling-near-miss)
  :precond ((w-topic-is mechanism)
            (w-variable-is ?vbl)
            (is-neural ?vbl))
  :recipe  ((retry-at (and (w-topic-is mechanism)
                           (w-level-is topic))))
  :hiercx  ())
```

Operator T7:

```
(def-operator tutor-by-elicit
  :goal (did-tutor ?topic ?vbl)
  :precond ((not (i-input-catg-is ling-near-miss))
            ...)
  :recipe  ((goal (did-elicit ?topic ?vbl)))
  :hiercx  ((assert (w-topic-is ?topic))
            (assert (w-level-is topic))))
```

Figure 2: Sample discourse operators

the exposition. Relations starting with *w-* are "working storage," i.e. facts added to the transient knowledge base to maintain state between turns. Relations starting with *p-* or with no prefix represent rhetorical knowledge and domain knowledge, respectively, in the permanent knowledge base.

Facts can be added and deleted from the transient knowledge base whenever desired using *assert* and *retract*. Additionally, the *:hiercx* slot of the operator syntax exists to allow operators to make decisions according to the current discourse context. Items in the *:hiercx* slot are instantiated and kept in the transient database only as long as the operator which spawned them is on the agenda.

Finally, APE permits the user to declare *external relations*. These relations are used in the same way as other knowledge base relations, but they do not actually appear in the knowledge base. Instead, when an external relation is

Operator T8:

```
(def-operator tutor-deep-mechanism-by-elicit-cont
  :goal (did-tutor-deep mechanism ?from-vbl ?to-vbl t)
  :precond ()
  :recipe  ((goal (did-utter "And "))
            (goal (did-elicit mechanism ?from-vbl)))
  :hiercx  ((assert (w-level-is prompt))))
```

Operator T9:

```
(def-operator tutor-by-inform    ;; not used in example
  :goal (did-tutor ?topic ?vbl)
  :precond (...)
  :recipe  ((goal (did-inform ?topic ?vbl)))
  :hiercx  ((assert (w-topic-is ?topic))
            (assert (w-level-is topic))))
```

Operator T10:

```
(def-operator tutor-by-restate
  :goal (did-tutor mechanism ?vbl)
  :precond ((is-neural ?vbl)
            (i-input-catg-is ling-near-miss))
  :recipe  ((goal (did-utter "In other words, the nervous system.")))
  :hiercx  ((assert (w-topic-is mechanism))
            (assert (w-level-is topic))))
```

Operator T11:

```
(def-operator elicit-mechanism-default
  :goal (did-elicit ?topic ?vbl)
  :precond ((w-topic-is mechanism))
  :recipe  ((prim-interactive (ask '("What controls" ?vbl "?"))))
  :hiercx  ())
```

Operator T12:

```
(def-operator utter
  :goal (did-utter ?utterance)
  :precond ()
  :recipe  ((primitive (say ?utterance)))
  :hiercx  ())
```

Figure 2 (continued): Sample discourse operators

called, a user-written function is requested to return a binding list with all possible sets of bindings, in the same format used by the knowledge base interface. The use of external relations allows the user to express preconditions which may be inconvenient to express in first-order logic, to connect to an external database, or to communicate with another program, such as a domain expert or a graphical user interface. In the examples below, names starting with *e-* represent external relations.

Communicating with Other Agents

The planner communicates with other processes via the transient knowledge base. Functions are provided for other processes to read and write facts in the knowledge base. In the example below, the user interface is handled by a process running in parallel with the planner. Relations starting with *i-* are added to the knowledge base by this process.

For responding to the user, the example below contains text generated directly by the planner. Forms representing text could also be sent to a text realization component whose interface is an external relation. Additionally, both text and graphical output could be sent to a GUI.

Generating a Dialogue

In this section we walk through an example to show how the APE planner can produce more complex dialogues than other dialogue systems. For this purpose we have reimplemented a piece of CIRCSIM-Tutor v. 2 (Woo et al. 1991; Zhou et al. 1999), a text-based ITS in physiology.

Figure 1 shows an example of text that can be generated with APE. It contains the text of one tutoring episode, i.e. it teaches the student one piece of domain knowledge. Following Grosz (1977), each tutoring episode is seen to be a subdialogue of the complete tutoring session. Although the generated text is real, it should be noted that the dialogue is not realistic in the sense that genuine students do not usually make so many consecutive distinct errors.

In this episode, the content to be communicated is the fact that total peripheral resistance or TPR, an important factor in the physiology of blood pressure, is under nervous system control. Figure 3 shows the relationship between TPR, the nervous system and other relevant physiological parameters. The arrow represents the relation "directly controls." The italicized parameters represent deeper knowledge that the student does not need to invoke in order to solve the problem. However, if the student does invoke that knowledge, the teacher wants to build on that information to help the student understand the role of the nervous system.

Turn 1 of Figure 1 is generated from the goal *(did-tutor mechanism TPR)*. One way to satisfy this goal is through operator T7. Operator T11 is one of a number of lower-level realization operators that generate different surface forms as realizations of the *did-elicit* subgoal. Operator T7 also asserts that the current topic is *mechanism*, and the current hierarchical discourse level is *topic*. These facts will be used to choose appropriate specializations of goals in later turns.

The correct answer to the question in turn 1 is the nervous system. Although the student does not give that answer in turn 2, the student gives an answer that the tutor can build on in a collaborative fashion. We sometimes call an answer such as this, one that is not the desired answer but not wrong either, a *near miss*. Near misses are a subcategory of the category Woolf (1984) labels "grain of truth" answers.

Assume that a plan operator not shown, one of a series of input-handling operators, cause the goal *did-handle-near-miss* to be invoked whenever the input processor recognizes a near miss. (Note that "near miss" is as much a pragmatic distinction as a semantic one.) In addition to examining the input, the preconditions of operator T1 use the previously stored topic and level information to restrict this operator to cases where its output will be appropriate in context.

For the first near miss in a subdialogue, operator T1 will always trigger operator T2. For the first near miss, the relation *w-max-nearmiss-is*, which keeps track of how close the student has come to the correct answer, has not yet been used. As a result, the *not*-clause in operator T2 is satisfied. The user-written external relation *e-on-leg* checks the data structure in Figure 3 to make sure that the student is following the arrows in the correct direction. If all the preconditions are satisfied, operator T2 will trigger one of a number of lower-level operators, such as operator T8. Other variants of operator T8 generate discourse markers that fit better into other contexts. The resulting output is seen in turn 3.

Assume that this time the student replies with a synonym of the previous answer, such as "radius of arterioles," as shown in turn 4. Operator T3 applies, followed by operator T12, causing turn 5 to be generated.

Now assume that the student replies "arteriolar diameter," as shown in turn 6. Arteriolar diameter is related to arteriolar radius, but the relationship is not causal. Operator T4, not shown because of its structural similarity to T3, applies in this case, causing turn 7 to be generated. A similar process is used if the student starts to go in the wrong direction, as shown in turn 8. In that case, operator T5, which is also not shown, applies, generating the text shown in turn 9.

Finally, assume that the student gives the right answer but not with the desired terminology, as shown in turn 10. This case, the "linguistic near miss," is handled by operator T6. This operator recognizes that the student's answer is essentially correct. Thus it uses *retry-at* to peel off any open goals which have accumulated since the topic and level were set at the beginning of the episode. The original goal is now on top of the agenda again, but the state of the world has changed so that operator T10 applies instead of T7, causing turn 11 to appear.

If the student had given a perfect answer, e.g. "nervous system," the system would have chosen an operator with a null recipe, causing the planner to go on to the next sibling (if one exists) or parent of the last goal. In the example, at this point the original goal is satisfied, the agenda is empty,

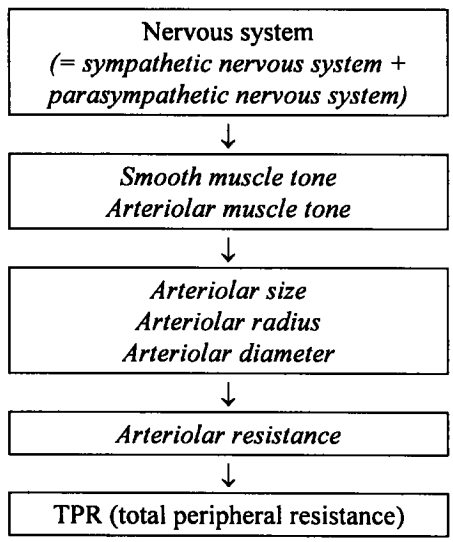

Figure 3: Domain knowledge

and the subdialogue is complete.

Operator T2 updated *w-max-nearmiss-is* to show how much progress the student made toward the correct answer on the first attempt. A similar operator, not shown, updates this relation when the student makes further progress. Note that operators 3, 4 and 5 do not update it because the student has not made further progress. For that reason, the question in turns 3, 5, 7 and 9 remains the same.

The relation *w-level-is prompt* is added to the transient knowledge base by the expansion of the *did-tutor-deep* goal that generates the question at the end of turns 3, 5, 7 and 9. When operator T3, T4 or T5 is invoked, the previous prompt is peeled from the agenda to prevent multiple prompts from becoming part of the tutor's model of the dialogue. This models the intuition that each of these prompts replaces its predecessor as an attempt to help the student answer the original question. If the student were to give a correct intermediate answer to one of the prompts, we would expect the tutor to build on that answer, not to return to an earlier prompt. If we did not remove the superfluous prompts from the agenda, the system would expect the student to answer all of the stacked-up prompts.

This example shows that APE can generate more complex dialogues than the system built by Zhou et al. (1999) to handle similar tutoring episodes. While Zhou's system could be described as an augmented finite state machine, APE allows arbitrarily deep nesting of discourse constructs. Although the finite-state machine approach can handle many types of near misses, it cannot handle nested ones, such as a linguistic correction to a near miss. More generally, many of the constructs in this example depend on nested goals and could not be generated using earlier approaches.

Conclusion

In this paper we have described APE, a reactive planner that we have implemented as part of the Atlas dialogue manager. It can be used to implement any conversation-based system, where turns in the "conversation" may include graphical actions and/or text. We have shown how this system can be used to generate a dialogue involving multiple types of answers to a question, each of which requires a different continuation. This system is intended largely to model dialogues with hierarchical, multi-turn plans. However, we have shown that it can be used as a general-purpose programming tool for implementing a dialogue system, since it can express the fundamental building blocks of sequence, selection and iteration, as identified by Dijkstra (1972).

Acknowledgments

This paper has been improved by comments from Mark Core, Abigail Gertner, Michael Glass, Neil Heffernan, Pamela Jordan, Bruce Mills, Martha Pollack, Michael Ringenberg, Kurt VanLehn, Yujian Zhou and the referees.

References

Dijkstra, E. W. 1972. Notes on Structured Programming. In Dahl, O.-J., Dijkstra, E. W. and Hoare, C. A. R., *Structured Programming*. London: Academic Press.

Erol, K., Hendler, J. and Nau, D. S. 1994. HTN Planning: Complexity and Expressivity. In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI '94), Seattle, WA.

Freedman, R. 1998. Atlas: A Plan Manager for Mixed-Initiative, Multimodal Dialogue. In AAAI '99 Workshop on Mixed-Initiative Intelligence, Orlando.

Freedman, R. and Evens, M. W. 1996. Generating and Revising Hierarchical Multi-Turn Text Plans in an ITS. In Intelligent Tutoring Systems: Third International Conference (ITS '96), Montreal, 632–640. Berlin: Springer. Lecture Notes in Computer Science 1086.

Georgeff, M. P. and Ingrand, F. F. 1989. Decision-Making in an Embedded Reasoning System. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI '89), Detroit, MI, 972–978.

Grosz, B. J. 1977. The Representation and Use of Focus in a System for Understanding Dialogs. In Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI '77), Cambridge, MA, 67–76.

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

Woo, C., Evens, M., Michael, J., Rovick, A. 1991. Dynamic Instructional Planning for an Intelligent Physiology Tutoring System. In Proceedings of the Fourth Annual IEEE Computer-Based Medical Systems Symposium, Baltimore, 226–233. Los Alamitos: IEEE Computer Society Press.

Woolf, B. 1984. Context-Dependent Planning in a Machine Tutor. Ph.D. diss., Dept. of Computer and Information Science, University of Massachusetts at Amherst. COINS Technical Report 84–21.

Yang, Q. 1990. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence* 6(1): 12–24.

Zhou, Y., Freedman, R., Glass, M., Michael, J. A., Rovick, A. A. and Evens, M. W. 1999. What Should the Tutor Do When the Student Cannot Answer a Question?. In Proceedings of the Twelfth Florida Artificial Intelligence Symposium (FLAIRS '99), Orlando, FL.