MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

A Plug-in Architecture for Generating Collaborative Agent Responses

Charles Rich, Neal Lesh, Jeff Rickel, Andrew Garland

TR2002-10 March 2002

Abstract

We describe an implemented architecture for programming the responsesto its developers.

Autonomous Agents and Multi-Agent Systems, July 2002. Bologna, Italy.

Submitted in November 2001; revised and released February 2002.

A Plug-in Architecture for Generating Collaborative Agent Responses

Charles Rich[*], Neal Lesh[*], Jeff Rickel[†] and Andrew Garland[*]

[*] Mitsubishi Electric Research Laboratories
201 Broadway
Cambridge, MA, 02139, USA
rich, lesh, garland@merl.com

[†] USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA, 90292, USA
rickel@isi.edu

ABSTRACT

We describe an implemented architecture for programming the responses of collaborative interface agents out of easily composable and reusable plug-in components, and discuss the underlying theoretical and practical issues. The power of the architecture comes primarily from a rich representation of collaborative discourse state, which includes a focus stack and plan tree. The architecture also provides a useful separation between the principles and preferences underlying an agent's behavior. We illustrate the use of plug-ins in a complex tutoring agent, which includes plug-ins that diagnose incorrect actions and explain why a step needs to be done. Plug-ins are part of the COLLAGEN agent-building middleware, which has been used by a number of researchers in addition to its developers.

Categories and Subject Descriptors

I.2 [**Computing Methodologies**]: Artificial Intelligence

General Terms

Algorithms, Design

Keywords

Interface agents, conversational agents, action selection and planning, agent architectures

1. INTRODUCTION

This paper addresses the problem of how to build collaborative interface agents out of composable and reusable behavioral components. Although this problem is fundamental to the engineering of collaborative interface agents, we did not fully appreciate it ourselves until we began to scale up to agents with a fairly high degree of behavioral complexity. This research takes place within the context of a multi-year, multi-person project called COLLAGEN (for COLLaborative AGENt) [11, 12]. COLLAGEN is also the name of our Java toolkit for building collaborative interface agents.

AAMAS'02, July 15-19, 2002, Bologna, Italy.

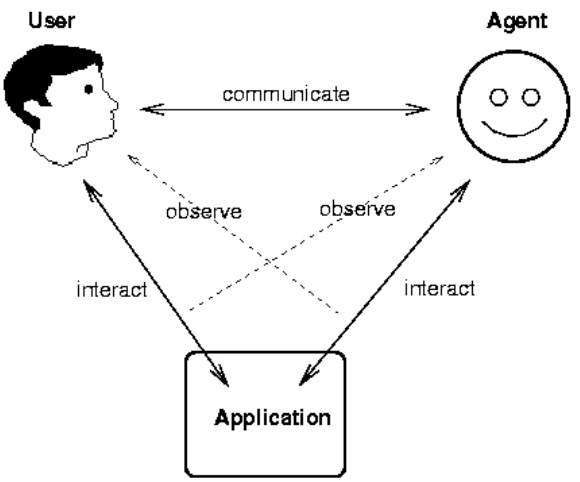

Figure 1: Collaborative interface agent paradigm.

1.1 Collaborative Interface Agents

A key practical goal of COLLAGEN is to make it possible to build collaborative interface agents with a maximum degree of software reuse between applications. Figure 1 illustrates what we mean by the term *collaborative interface agent*. In this paradigm, a software agent plays the same role that one of two humans plays when two humans collaborate on a task involving a shared artifact, such as two mechanics working on a car engine together or two computer users working on a spreadsheet together.

In general, *collaboration* is defined as a process in which two or more participants coordinate their actions toward achieving shared goals. In this paper, and in all of our work on COLLAGEN, we focus exclusively on collaborations involving only two participants. Collaboration usually requires some form of communication between the participants, typically in natural language, but also sometimes using gestures and specialized or artificial languages. Notice in Figure 1 that, in addition to communicating with each other, both the user and the software agent can interact with the shared artifact (typically a software application) and observe each other's interactions with the shared artifact.

Collaboration is a very broad concept, covering a wide spectrum of interactions, depending on the relative expertise and initiative of the participants and on the primary shared goal of the collaboration [3]. For example, tutoring

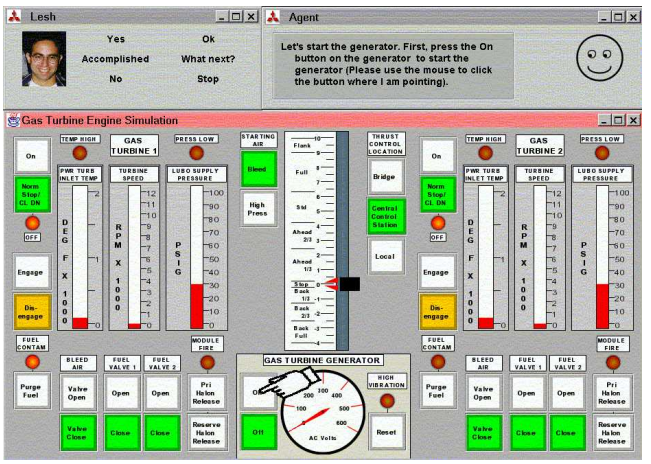

Figure 2: Intelligent tutor for a gas turbine engine.

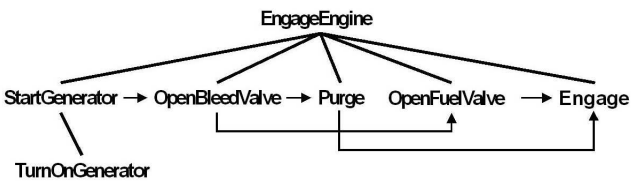

Figure 3: Fragment of gas turbine task model.

is a kind of collaboration in which one participant (the tutor) has much greater expertise and initiative, and the primary shared goal is to increase the student's expertise. At the other end of the spectrum, assistance is a kind of collaboration in which one participant (the assistant) has somewhat less expertise and initiative, and the primary shared goal is to successfully accomplish some task. In the middle of the spectrum are peer collaborations, in which the expertise and initiative of the participants are relatively evenly matched.

1.2 A Complex Agent

To motivate and illustrate the points of this paper, we introduce a COLLAGEN-based tutoring agent [14, 15], which teaches the standard operating procedures for a gas turbine engine using a software simulator.

This system is an instantiation of the paradigm in Figure 1, in which the shared application is the software simulator shown in the large lower window of the screen shot in Figure 2. The smaller window in the upper right corner of Figure 2 displays the agent's verbal communication to the human student; the window in the upper left corner is used by the student to verbally communicate to the agent. (In this paper, the student only selects one of six simple utterances; we have also developed a more powerful spoken-language interface.) Both the agent and the student can also point and click on the simulator interface. Figure 2 shows the agent pointing at the "generator on" button.

An example collaboration between this tutoring agent and a student, shown in Figure 4, illustrates the variety of collaborative responses of which this agent is capable, including:

- teaching the next step in a procedure (4,13,23,33,38)
- confirming successful completion of a subtask (8,37,42)
- turning over initiative to the student (15)
- re-focusing the student on the current subtask (29)

1. Tutor says "Our first exercise is very simple." NEX EXE
2. Tutor sets the state of the simulator. NEX EXE
3. Tutor says "Let's start the generator." PSJ
4. Tutor says "The first step is to press the generator on button. TSP
5. Tutor says "Please press the generator on button. PSU
6. Student presses the generator on button.
7. Tutor says "Great." PFB
8. Tutor says "We confirm successfully starting the generator by noticing a positive reading on the volt meter." TCF
9. Tutor says "Now, do you want to try a more complex task that includes starting the generator?" NEX EXE
10. Student says "Yes."
11. Tutor sets the state of the simulator. NEX EXE
12. Tutor says "Let's engage engine one." PSJ
13. Tutor says "The first step is to start the generator." TSP
14. Tutor says "Let's start the generator." PSJ
15. Tutor says "You take it from here." TIN
16. Student presses the disengage button.
17. Tutor says "Please stop disengaging the engine." INT
18. Student says "What next?"
19. Tutor says "Please press the generator on button." PSU
20. Student presses the generator on button.
21. Tutor says "Great." PFB
22. Tutor says "These engines need a supply of compressed air in order to be started." TXP
23. Tutor says "The next step is to open the bleed valve to tap into the air produced by the generator." TSP
24. Tutor says "Please open the bleed valve." PSU
25. Student presses the engage button.
26. Tutor says "Whoops, it's too soon to do that." PRE
27. Student opens fuel valve one.
28. Tutor says "That was a correct action, but I wasn't expecting you to do it now." UNL
29. Tutor says "Let's go back to opening the bleed valve." PVF
30. Student opens the bleed valve.
31. Tutor says "Right." PFB
32. Tutor says "Now, notice that the fuel contamination light is on." TAP
33. Tutor says "The next step is to press the purge button to clear fuel line contamination." TSP
34. Tutor says "Please press the purge button." PSU
35. Student presses the purge button.
36. Tutor says "Great." PFB
37. Tutor says "We confirm successfully clearing fuel line contamination by noticing that the fuel contamination light is off." TCF
38. Tutor says "The next step is to press the engage button again." TSP
39. Tutor says "Please press the engage button." PSU
40. Student presses the engage button.
41. Tutor says "Great." PFB
42. Tutor says "We succeeded in engaging engine one." TSU

Figure 4: Example tutoring collaboration.

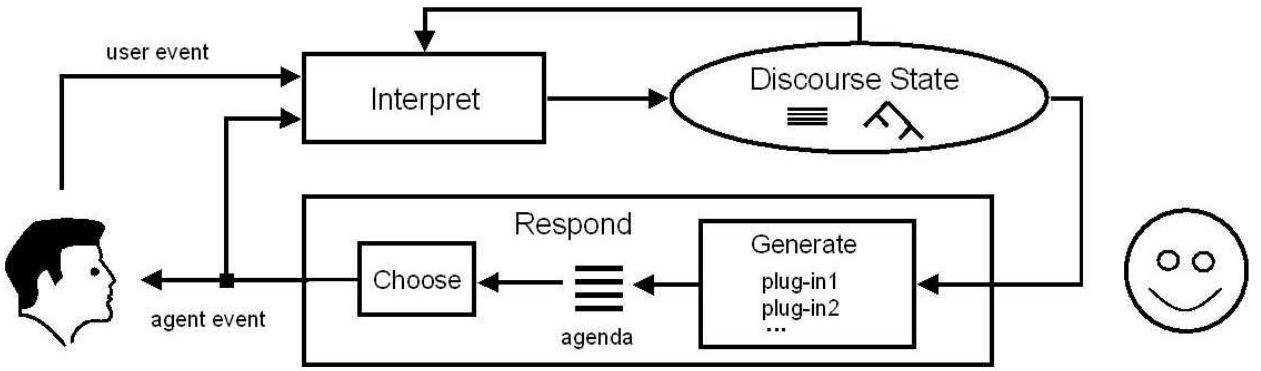

Figure 5: High-level agent response architecture.

- providing positive feedback (7,21,31,36,41)
- diagnosing and critiquing incorrect actions (26,28)
- explaining why a step needs to be done (22)

Many of these behaviors are not limited to tutoring, but would also be useful in a wide range of collaborations.

The numbers in parentheses above refer to lines of Figure 4. The three-letter codes in Figure 4 identify the plug-ins which produced each line (described later). Figure 3 will be discussed in Section 4.

1.3 The Problem

At a very high level, the problem we address in this paper is how to build, evolve, and debug a complex agent like the one in Section 1.2 using behavioral components that we can easily combine and reuse ("mix and match"). More specifically, we want to solve this problem within a principled architecture which embodies collaborative discourse theory [5, 10].

A not-so-straw man solution to the high-level problem might be to use traditional situation-action rules. For example, you might imagine producing the agent's responses on lines 21–24 of Figure 4 with a rule along the lines of:

> *If* the student presses the generator on button, *then* say "... Please open the bleed value."

This approach is not entirely a straw man for two reasons. First, many simple agents are in fact built this way. Second, some of the basic intuitions underlying the rule-based approach are sound. For example, including or not including a particular rule is an easy way of adjusting the behavior of a given agent. Furthermore, if you factor out the shared task model (as we describe in Section 2.1 below), you can rewrite such rules in a more application-independent form, such as:

> *If* the student successfully completed a step in the current procedure, *then* teach the next step.

A rule of this form corresponds to an easily describable, reusable component of behavior, such as "teaching the next step in a procedure" and the other examples listed at the end of Section 1.2. What this illustrates is that the key issue in achieving modularity and reuse is the *representation* of the situation against which a rule matches.

Our reusable behavioral components, which we call *plug-ins*, have some rule-like aspects. What makes our plug-ins unique, however, is how they are invoked with respect to a rich and abstract situation representation based on collaborative discourse theory.

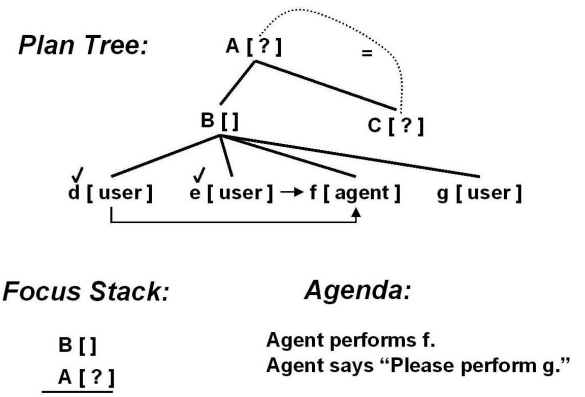

Figure 6: Example discourse state and agenda.

2. SYSTEM ARCHITECTURE

Figure 5 gives a high-level view of COLLAGEN's agent response architecture. The heart of this architecture is the representation of discourse state. Whenever a new event occurs, the first step is to *interpret* the event with respect to the current discourse state, resulting in an updated discourse state. Then, how the agent will *respond* is computed from the updated discourse state.

2.1 Discourse State

Discourse is a technical term for an extended communication between two or more participants in a shared context, such as a collaboration. In previous work [11, 12], we have discussed at length the crucial role which collaborative discourse theory [5, 10] has played in the development and architecture of COLLAGEN. Our discourse state representation consists of a stack of goals,[1] called the *focus stack*, and a *plan tree* for each goal on the stack. The top goal on the focus stack is the current task or subtask. The plan tree associated with each goal gives its hierarchical decomposition, if any, into partially ordered sets of subtasks with constraints.

Figure 6 shows a discourse state involving a very simple task model, which we will use below to illustrate COLLAGEN's algorithm for invoking plug-ins. (It also shows the agenda generated from this state by the default plug-ins, as discussed in Sections 2.3 and 3.3.)

A, B and C are non-primitive actions; d, e, f and g are

[1]To support natural language understanding, each element of the stack is in fact a focus space [5], which includes a goal.

primitive actions. Primitive actions include both "physical" actions, such as pressing a button, and utterances. All actions may have parameters (indicated in square brackets), which may be bound or unbound (indicated by ?). The implicit first parameter of every primitive action is which participant (i.e., the user or the agent) should or did execute it.A and C have single parameters which are constrained in this plan to be identical.

Action A has been decomposed into B and C (with no ordering constraint). B has been decomposed into d, e, f and g with the ordering constraints indicated by arrows. Primitive actions d and e have already been executed (indicated by check marks). C has not yet been decomposed.

Not shown in this figure are the goal decomposition rules, called *recipes*. In general, there are one or more recipes for each non-primitive action type, which may be chosen based on the recipe's applicability conditions. A collection of action types and recipes constitutes COLLAGEN's explicit task model. The task model can be thought of as specifying possible ways of achieving tasks in the domain. The discourse state plan tree is an instantiation of parts of the task model under the present conditions.

The focus stack in Figure 6 contains two elements. The current task (top of the stack) is B. When B is achieved, we expect to continue working on A. Note that in the presence of interruptions and unexpected focus shifts (Section 3.1.2), the element underneath each goal in the stack is not always the goal's plan tree parent.

2.2 Discourse Interpretation

Most of our research and writing on COLLAGEN has concentrated on the *interpret* step in Figure 5. In contrast, this work is primarily about the *respond* step. However, to understand the operation of the respond step, we first need to briefly summarize discourse interpretation.

The basic job of discourse interpretation is to explain how the current event (primitive action) contributes to a goal in the current plan tree or starts a new goal in the task model. An utterance typically contributes to a goal by satisfying a knowledge precondition [10], such as asking about or providing the value for an unknown parameter. A non-utterance typically contributes to a goal by matching one of the steps in the plan or a subplan for that goal. (Utterances can also be explicit plan steps.) Plan recognition [8] plays a central role in the discourse interpretation algorithm.

Updating the discourse state typically involves some combination of extending the plan tree and popping and/or pushing the focus stack. Note that COLLAGEN also applies the same interpretation and update algorithm to the agent's responses, so that the discourse state reflects the mutual beliefs[2] of the agent and user.

2.3 Response Production

The *respond* step in Figure 5 has two stages (both of which will be described in more detail in the next section). First, the current set of plug-ins is invoked to generate a list, called the *agenda*, of candidate primitive actions (including utterances). Then, one of these candidates is chosen as the actual response.

Intuitively, the algorithm for generating the agenda is the "inverse" of the discourse interpretation algorithm, in the sense that discourse interpretation determines how a given primitive action contributes to the current collaboration, whereas discourse generation computes a set of primitive actions which *would* contribute to the current collaboration.

2.4 Principles and Preferences

The decomposition of agent response production in Figure 5 into a *generate* and a *choose* step corresponds to a useful methodological division, which we term "principles and preferences." The basic idea is that all of the items in the agenda should be reasonable responses in principle; which item is chosen should be only a matter of preference depending, for example, on the personality of the agent.

For example, it would *not* be reasonable in principle for the agent to propose executing an action which had just been successfully executed, or to ask for the value of a parameter whose value has already been agreed upon. Therefore, these utterances should never appear in the agenda under these circumstances.

On the other hand, if there are two unbound parameters in the current task, choosing which to ask about first is a matter of preference. Similarly, if there are two primitive actions which could be executed next, one of which must be executed by the agent and one of which must be executed by the user, which action the agent deals with next would be a preference.

Although the distinction between principles and preferences is not always sharp, we have found it very useful for two reasons. First, it helps us think about the correctness of a plug-in independent of other plug-ins, i.e., all the candidates generated by a plug-in should be reasonable in principle in the given circumstance, even if some other utterance or action might be preferred.

Second, we have found it much easier to debug the current plug-in version of the tutoring agent in Section 1.2 than an earlier version which, instead of plug-ins, had a complex conditionalized body of code to decide what to do next. When the current agent does the wrong thing at some point in a scenario, we first ask the question: Was the "right" behavior in the agenda at that point? If so, we can then focus on why the right behavior was not chosen. If not, we can focus on debugging the plug-in that we expected to generate the right behavior.

3. PLUG-INS

A plug-in is an "object" in the sense of object-oriented programming, i.e., each plug-in is an instance of a type, may contain private state, and implements the following methods (*event* is the primitive action which just occurred; *node* is a plan tree node):

update(*event*) Update the private state of the plug-in (if any) and modify the discourse state[3] (if necessary), e.g., by adding a goal or binding a parameter. This method is executed entirely for its side effects.

[2]Technically, the discourse state represents the agent's beliefs about the mutual beliefs of the agent and user. We are currently extending COLLAGEN to reuse some of the same mechanisms to represent the agent's private beliefs (including private plans) and the agent's beliefs about the user's beliefs, i.e., what is called the "student model" in tutoring applications.

[3]A plug-in should really modify only the agent's private beliefs, which will be supported in a future COLLAGEN release.

generate(*event*) Return a (possibly empty) list of agenda items which would be an immediate response to *event*. This method should have no side effects.

visit(*node*) Return a (possibly empty) list of agenda items which would contribute to the goal of *node*. This method should have no side effects.

3.1 Generating Agenda Items

The algorithm for invoking plug-ins to generate agenda items has three steps:

1. Invoke the *update* method of each plug-in.
2. Invoke the *generate* method of each plug-in.
3. Invoke the *visit* method of each plug-in on each live, expected node in the current discourse state.

The combined results of invoking the *generate* and *visit* methods form the agenda of agent response candidates.

3.1.1 Live Nodes

A plan tree node is *live* if and only if that node's action (primitive or non-primitive) has not already been achieved, its precondition is not false, all of its temporally constrained predecessors have been achieved, and none of its temporally constrained successors have been started.

Note that the final clause in the liveness condition above is due to the presence of *optional* steps. Unachieved optional steps do not prevent their successors from being live, but should not be live themselves once they have been skipped. The presence of optional steps also introduces some subtlety into the plan tree traversal algorithm in Section 3.2.1, since a plan is considered complete when all of its non-optional steps are achieved (even if some optional steps remain unachieved).

3.1.2 Expected Nodes

The definition of expected nodes comes from the discourse interpretation algorithm, via the concept of focus shift.

A *focus shift* occurs whenever discourse interpretation modifies the focus stack. Focus shifts that involve popping finished tasks are *expected*; focus shifts that involve starting to work on a different task before the current task is finished are *unexpected* [9]. (Note that finished tasks remain on the stack until the interpretation of the next event, because a just-completed goal may continue to be the topic of current conversation.)

Slightly more technically, an unexpected focus shift occurs iff an unachieved current goal (the top of the stack) is either popped off the stack or an *interruption* (a new goal which does not contribute to the current goal) is pushed on top of it. All other focus shifts are expected. For example, given the discourse state in Figure 6, saying "Let's achieve C" would be an unexpected focus shift, because its interpretation involves popping the unachieved goal B off the stack.

By induction, we define a plan tree node as *expected* iff its goal is either already the current task (top of the stack) or starting to work on its goal would be interpreted as an expected focus shift. Note that not all live nodes are expected. For example, in Figure 6, C is live but not expected.

By visiting only expected nodes, COLLAGEN's response generation algorithm provides a built-in bias towards focused behavior [9] on the part of the agent. If, as recommended, all the agenda items returned by a *visit* method contribute to the given node, choosing one of these items cannot lead to an unexpected focus shift. Plug-ins, such as PrivateFocus in Section 4, which are intended to make unexpected focus shifts, should use the *generate* method.

3.2 Choosing a Response

The main issue in choosing a response from the generated agenda items is how to express the preferences. We have taken a pragmatic approach, using a combination of default ordering and explicit preferences. The basic idea is to visit the live, expected nodes (step 3 in the algorithm above) in a well-chosen default order, do a post-pass to reorder the agenda based on any additional explicit preferences, and then choose the first item on the agenda as the response. We provide the details below.

3.2.1 Default Order

First we assume that the plug-ins themselves have a default order (e.g., the order in which they are added to the system). The agenda is initialized by adding the results, in the order returned, of invoking the *generate* method of each plug-in, in the default plug-in order.

The initial order of the remaining items in the agenda is determined by the order in which the live, expected nodes are visited. At each node, we add to the agenda the results, in the order returned, of invoking the *visit* method of each plug-in on that node, in the default plug-in order. The following algorithm specifies the order in which the nodes are visited:

1. Do a pre-order breadth-first traversal starting at the node corresponding to the top of the focus stack. Visit only live nodes and do not descend into subtrees whose root is not live (achieved nodes are not live).
2. If the starting node of 1 has been achieved, repeat 1 and 2 starting with the stack element one below in the stack, if any (repeat as many times as necessary).

For example, this algorithm visits the plan tree nodes shown in Figure 6 in the order: B, f, g. In our experience, pre-order breadth-first traversal is a good default because it causes the agent to work on tasks in a top-down style. The repeated "popping" of achieved goals off the stack in step 2 is required to visit all the expected nodes.

3.2.2 Explicit Preferences

We have experimented with expressing explicit preferences using priorities (attached to each agenda item by the plug-in that produced it) and using pairwise comparison rules, as in SOAR [7]. Comparison rules are more expressive, but priorities are more convenient. The tutoring agent described in Section 1.2 was implemented using priorities, which are described in Section 4.

3.3 Default Plug-ins

Part of the plug-in metaphor is the idea that you don't have to write them all yourself, but can obtain them from other sources. In this section, we describe COLLAGEN's six default plug-ins, which implement a normative collaboration style, somewhere in the middle of the collaboration spectrum closer to the intelligent assistant end. Section 4 describes in less detail a larger collection of plug-ins which we developed for the tutoring agent in Section 1.2.

User performs d.	
User performs e.	
User says "Ok, what next?"	
Agent performs f.	EXE
Agent says "Please perform g."	PSU
User performs g.	
...	
User says "Please perform d."	
Agent performs d.	EXE
Agent says "Who should perform e?"	AWO
User says "You should perform e."	
Agent performs e.	EXE
Agent performs f.	EXE
Agent says "Please perform g."	PSU
...	
User says "Let's achieve A."	
Agent says "What is the parameter of A?"	AWT
User says "The parameter of A is 5."	
Agent says "Let's achieve B."	PSJ
User says "Ok."	
Agent says "Who should perform d?"	AWO

Figure 7: Three of many possible collaborations generated by the default plug-ins.

Below we give an informal word definition of each default plug-in's *visit* method. None of these plug-ins use their *update* or *generate* methods. Note that the names of the last five plug-ins derive from the semantic representation (see Section 3.4.2) of the utterance types they return. The variable *goal* refers to the primitive or non-primitive action associated with the visited node in the plan tree. (The three-letter codes in parentheses below refer to lines in Figure 7 resulting from the given plug-in.)

Execute (EXE) If *goal* is primitive and all of its parameters are bound and its first parameter is bound to the agent, then return *goal*.

ProposeShouldUser (PSU) If *goal* is primitive and its first parameter is bound to the user, then return an utterance of the form "Please perform/say *goal*."

ProposeShouldJoint (PSJ) If *goal* is non-primitive and it is not already on the focus stack, then return an utterance of the form "Let's achieve *goal*."

AskWho (AWO) If *goal* is primitive and its first parameter is unbound, then return an utterance of the form "Who should perform/say *goal*?"

AskWhat (AWT) For each unbound *parameter* in *goal* (other than the first parameter of primitive actions), return an utterance of the form "What is the *parameter* of *goal*?"

AskHow If *goal* is non-primitive and there is more than one applicable recipe for *goal*, then return an utterance of the form "How should we achieve *goal*?"

These particular plug-ins are COLLAGEN's default because they correspond directly to the basic cases, according to collaborative discourse theory [5, 10], for how an action can contribute to a goal. The first three plug-ins correspond to achieving the steps of a plan. (The first two plug-ins produce the two agenda items shown in Figure 6.) The last three plug-ins correspond to knowledge preconditions [10].

Combined with a rich task model, these six plug-ins can give rise to a surprising variety of collaborative agent responses, which automatically adapt to the user's behavior. In fact, most of the demonstration applications we have built use only these default plug-ins plus one or two small application-specific plug-ins.

Figure 7 shows three of literally dozens of collaborative variations that arise even with the very simple task model shown in Figure 6 (the plug-in responsible for each agent response is indicated by the three-letter code at the end of the line). The first interaction passes through the discourse state shown in Figure 6. In the second interaction, the user asks the agent to perform d and e.[4] The user initiates the third interaction by explicitly proposing the toplevel goal, which leads to a discussion of its parameter.

Finally, it is worth emphasizing that the robustness of these plug-ins is due in large part to the fact that their output depends only on the current discourse state, not on the current event. This is a desirable property to keep in mind when writing new plug-ins.

As a simple example, consider the fact that in the first interaction in Figure 7, the Execute plug-in would return the same agent response (to perform f) regardless of the order in which the user performed d and e. Both sequences of user events lead to the same state (pictured in Figure 6).

3.4 Other Issues

In this section, we briefly discuss two important issues which impinge upon the understanding of how plug-ins work, but whose full treatment is beyond the scope of this paper.

3.4.1 Turn Taking

One might get the impression from the term "agent response" that interaction between the user and agent always occurs in strict alternation, i.e., one user event followed by one agent event, and so on. In human-human collaboration, however, the rules of so-called "turn taking" are quite complex and are influenced by many real-time factors, including body posture, eye movement, and speech intonation. COLLAGEN has a separate layer of discourse processing that implements turn-taking.

Our first approximation to turn-taking rules are that a user's turn consists of zero or more physical actions followed by an utterance, and that the agent's turn consists of zero or more actions followed by one or more utterances (the agent decides when it is done speaking). For example, if the user executes an action, the agent will wait for a user utterance before responding.

Although they work relatively well in simple interactions, we are not satisfied with COLLAGEN's current turn-taking rules, because they require too much ad hoc tuning in more complex applications. We are currently working on a completely new approach to turn taking based on a deeper analysis of natural human behavior.

3.4.2 Natural Language Processing

COLLAGEN per se is not a complete natural-language processing system. In terms of the architecture of Figure 5, what

[4] In the starting plan for B (instantiated by a recipe for B) the first parameters of d and e were unbound. These parameters become bound when the primitive actions are executed.

COLLAGEN fundamentally provides is a mapping from the *semantic* (logical) representation of events (utterances and other primitive actions) to the *semantic* representation of the agent response. Internally, all of COLLAGEN's processing uses a semantic representation based on Sidner's artificial discourse language [16].

Thus, when the description of a plug-in, such as ProposeShouldJoint, says it returns an utterance of the form "Let's achieve *goal*," what is actually being returned is a semantic representation, such as Propose(agent, Should(*goal*)).

The mapping back and forth between the semantic representation and the surface representation of an utterance (e.g., English text or spoken language) is done by other components, which are used in concert with COLLAGEN. We have used different such components in different prototypes built with COLLAGEN. For the tutoring system of Section 1.2, we used IBM ViaVoice to map from speech to text, the Java Speech API to map from text to semantic representation, a homegrown template-driven algorithm to map from semantic representation to text, and IBM ViaVoice to map from text to speech.

4. A COMPLEX AGENT REVISITED

We return now to the tutorial agent introduced in Section 1.2, and describe all of the plug-ins which are needed (in addition to the default plug-ins of Section 3.3) to produce the agent responses in the Figure 2. Space does not allow us to describe each plug-in in detail; for some, we provide only an example of the form of utterance returned.

The number in bold following the names of plug-ins with *visit* or *generate* methods is the priority of the returned items (see Section 3.2); the default plug-ins all have priority zero. The three-letter codes in parentheses after plug-in names are used in Figure 2 to identify the lines which result from a given plug-in; these line numbers are also referenced in parentheses at the end of each plug-in definition. Plug-ins without codes or line numbers happen not to be used in Figure 2. Figure 3 shows a fragment of the underlying task model, which will aid in understanding how the plug-ins below are invoked.

The first group of plug-ins relate directly to pedagogical strategy. Like the default plug-ins, they only use the *visit* method. A goal is *teachable* below iff it is part of the task model, its parent is the top of the focus stack, and the student model indicates that the student does not already know it. Notice these plug-ins are listed in increasing priority.

TeachStep (TSP) 100 If *goal* is teachable, then return an utterance of the form "The first/next step is *goal*." (4, 13, 23, 33, 38)

TeachExplain (TXP) 110 If *goal* is teachable and there is an explanatory utterance associated with *goal* in the task model and the utterance has not already been produced, then return the explanatory utterance. (22)

TeachApplicable (TAP) 120 If *goal* is teachable and there is an *observable* (such as a gauge) which indicates that the goal is applicable and this fact has not already been taught, then return an utterance of the form "Now notice *observable*." (32)

TeachInitiative (TIN) 130 If the student model indicates that the student should already know how to achieve *goal*, then return "You take it from here." (15)

TeachSuccessful (TSU) 140 "We have succeeded in *goal*." (42)

TeachConfirm (TCF) 150 "We confirm successfully *goal* by noticing that *observable*." (8, 37)

The following plug-ins respond to correct (PositiveFeedback plug-in) and incorrect actions (the rest of the plug-ins) via the *generate* method (*action* is the action performed or proposed by the current event). Note that a single action can be incorrect for more than one reason.

PositiveFeedback (PFB) 160 If *action* was live and expected, then return an utterance such as "Great", "Nice", "Right". (7, 21, 31, 36, 41)

StopInterruption (INT) 170 If *action* is an interruption, then return an utterance of the form "Please stop *goal*," where *goal* is the root of the interruption plan tree. (17)

Predecessors (PRE) 180 If *action* has unachieved predecessors, then return "Whoops, it's too soon to do that." (26)

Precondition 190 If *action* has a false precondition, then return "Whoops, that's not appropriate in the current situation."

NearMiss 200 If *action* is the right type of action, but one or more of its parameters are wrong, then return "Whoops, you didn't do that exactly right."

Repeated 210 "Whoops, you already did that"

The following two plug-ins deal with unexpected focus shifts by the student. Note that the PrivateFocus plug-in has both an *update* and a *generate* method.

PrivateFocus (PVF) 220

update: Maintains a *private focus*, which is the goal in the current plan that the agent wants to work on. This allows the agent to remember what goal to return to when the student makes an unexpected focus shift.

generate: If the top of the stack is different than the private focus, then return an utterance of the form "Let's go back to *private focus*." (29)

UnexpectedLive (UNL) 230 If *action* is live but unexpected, then return "That was a correct action, but I wasn't expecting you to do it now." (28)

Finally, the following plug-in has only an *update* method, which adds nodes to the toplevel plan for the tutoring session to introduce the next exercise and initialize the simulator appropriately. These new nodes are subsequently returned by the default Execute plug, as indicated by the "NEX EXE" code in Figure 4.

NextExercise (NEX) Whenever the current exercise is complete, chooses the appropriate next exercise and adds nodes to the toplevel plan to start it. (1, 2, 9, 11)

There are many points during the interaction in Figure 4 at which the agenda has multiple candidates, which are decided between by priorities. To give one example, after line 20, the PositiveFeedback plug-in returns "Great," because the student has correctly pressed the generator on button.

In addition, because the next live step is to OpenBleedValue (see Figure 3), ProposeShouldUser returns the utterance on line 24, TeachStep returns the utterance on line 23, and TeachExplain returns utterance on line 22. All of these utterances are reasonable after line 20; the assigned priorities implement a one of many possible pedagogical and communication policies.

5. RELATED WORK

The key insight of using collaborative discourse theory to generate an agenda of possible contributions to the current goal is due to Lochbaum [10]. The decomposition of the generation process into plug-ins is unique to our work.

In relation to AI planning research, this works falls more within the realm of plan execution and monitoring, rather than plan generation. Most work on plan execution and monitoring [1] deals with issues of uncertainty and failure during execution, whereas we are primarily concerned with the communication about plans that is required for collaboration. Other work that also puts planning within a communication context [17, 18] has not addressed the scaling up of behavioral complexity that motivates our plug-in architecture.

Many general-purpose agent-building systems provide composable, reusable, rule-like chunks; the most sophisticated is probably SOAR [7]. None of these systems, however, embody a theory of collaborative discourse, which means that the agent developer needs to explicitly program the kind of collaborative behaviors that are the default when using COLLAGEN.

STEVE [13] has a very similar collaboration model to COLLAGEN's. COLLAGEN, however, has a more general discourse state representation and interpretation algorithm. In fact, it was the effort to reconstruct STEVE (not including the virtual reality aspects) as the COLLAGEN-based tutor in this paper which germinated the plug-in idea [14, 15].

6. CONCLUSION

We have described an implemented architecture for developing sophisticated collaborative interface agents and discussed the theoretical and practical issues underlying its design. Broadly speaking, there are two main conclusions which can be applied to the design of other similar systems.

First, the ultimate source of the modularity which is desirable in developing collaborative agents lies in the choice of state representation. Our representation of discourse state benefits from being based on a firm theoretical foundations.

Second, at a more practical level, the distinction between principles and preferences, even if it is sometimes a little fuzzy, is a great aid in debugging the interaction of composable, reusable behavior components.

Finally, to summarize the status of COLLAGEN, we and our collaborators have, in addition to the tutor described in this paper, built COLLAGEN-based intelligent assistants for air travel planning [11], email [6], a graphical interface development tool [12], a video cassette recorder [12], a programmable home thermostat [4], and an embedded training tutor for airport approach path design [2]. All of these agents are currently research prototypes. Only the approach path tutor and the tutor described in this paper use the full plug-in architecture—the other agents use an earlier version of the architecture which generates the agenda in a less modular way. The approach path tutor uses most, but not all, of the plug-ins described in this paper, plus a few more that were specially developed for it.

7. REFERENCES

[1] R. Bergmann and A. Kott. Integrating planning, scheduling and execution in dynamic and uncertain environments. AAAI Tech. Report WS-98-02.

[2] B. Cheikes and A. Gertner, Teaching to Plan and Planning to Teach in an Embedded Training System. In *Proc. 10th Int. Conf. on Artificial Intelligence in Education*, pages 398–409, San Antonio, TX, May 2001.

[3] J. Davies, N. Lesh, C. Rich, C. Sidner, A. Gertner, and J. Rickel. Incorporating tutorial strategies into an intelligent assistant. In *Proc. Int. Conf. on Intelligent User Interfaces*, pages 53–56, Santa Fe, NM, 2001.

[4] E. DeKoven, D. Keyson, and A. Freudenthal. Designing collaboration in consumer products. In *Proc. ACM Conf. on Computer Human Interaction, Extended Abstracts*, pages 195–196, Seattle, WA, 2001.

[5] B. J. Grosz and C. L. Sidner. Attention, intentions, and the structure of discourse. *Computational Linguistics*, 12(3):175–204, 1986.

[6] D. Gruen, C. Sidner, C. Boettner, and C. Rich. A collaborative assistant for email. In *Proc. ACM Conf. on Computer Human Interaction, Extended Abstracts*, pages 196–197, Pittsburgh, PA, 1999.

[7] J. Laird, A. Newell, and P. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

[8] N. Lesh, C. Rich, and C. Sidner. Using plan recognition in human-computer collaboration. In *Proc. 7th Int. Conf. on User Modelling*, pages 23–32, Banff, Canada, June 1999.

[9] N. Lesh, C. Rich, and C. Sidner. Collaborating with focused and unfocused users under imperfect communication. In *Proc. 9th Int. Conf. on User Modelling*, pages 64–73, Sonthofen, Germany, July 2001.

[10] K. E. Lochbaum. A collaborative planning model of intentional structure. *Computational Linguistics*, 24(4):525–572, Dec. 1998.

[11] C. Rich and C. Sidner. Collagen: A collaboration manager for software interface agents. *User Modeling and User-Adapted Interaction*, 8(3/4):315–350, 1998.

[12] C. Rich, C. Sidner, and N. Lesh. Collagen: Applying collaborative discourse theory to human-computer interaction. *AI Magazine*, 22(4), 2001.

[13] J. Rickel and W. L. Johnson. Task-oriented collaboration with embodied agents in virtual worlds. In J. Cassell, J. Sullivan, and S. Prevost, editors, *Embodied Conversational Agents*, pages 95–122. MIT Press, 2000.

[14] J. Rickel, N. Lesh, C. Rich, C. Sidner, and A. Gertner. Building a bridge between intelligent tutoring and collaborative dialogue systems. In *Proc. 10th Int. Conf. on Artificial Intelligence in Education*, pages 592–594, San Antonio, TX, May 2001.

[15] J. Rickel, N. Lesh, C. Rich, C. Sidner, and A. Gertner. Collaborative discourse theory as a foundation for tutorial dialogue. In *6th Int. Conf. on Intelligent Tutoring Systems*, Biarritz, France, June 2002.

[16] C. L. Sidner. An artificial discourse language for collaborative negotiation. In *Proc. 12th National Conf. on AI*, pages 814–819, Seattle, WA, 1994.

[17] M. Tambe. Towards flexible teamwork. *J. of Artificial Intelligence Research*, 7:83–124, 1997.

[18] R. M. Young, J. Moore, and M. Pollack. Towards a principled representation for discourse plans. In *Proc. 16th Annual Conf. of the Cognitive Science Society*, pages 946–951, Hillsdale, NJ, 1994.