Using an Activity Model to Address Issues in Task-Oriented Dialogue Interaction over Extended Periods

Alexander Gruenstein and Lawrence Cavedon
Center for the Study of Language and Information
Stanford University, CA 94305
{alexgru,lcavedon}@csli.stanford.edu

Introduction

Collaborative natural dialogue offers a powerful medium for interaction between humans and intelligent autonomous agents (Allen *et al.* 2001). This is particularly the case for complex systems operating in dynamic, real-time environments, containing multiple concurrent activities and events which may succeed, fail, become cancelled or revised, or otherwise warrant discussion. Human dialogue in such scenarios is highly collaborative (Clark 1996), with much negotiation between instructor and task-performer: dialogue is used to specify and clarify goals and tasks, to monitor progress, and to negotiate the joint solution of any problems. Further, an interface to a device operating in such conditions must be interruptible, context-dependent, and otherwise extremely amenable to multiple threads of conversation.

Dealing with interaction over an extended period—we envisage dialogues over a period of minutes to hours—raises specific challenges for systems operating in complex, dynamic environments. This is particularly the case when a user is operating multiple devices, requiring attention-switching between devices and interactions, and long gaps between interactions with a specific device. In such circumstances, some of the specific challenges for task-oriented dialogue systems are the following:

- mixed initiative: the autonomous system must be able to raise issues or initiate a conversation thread based on its observations or the execution-status of its activities;
- task-reporting: the agent must report progress on the performance of its activities, to a level of relevance and detail that is appropriate;
- explanation: the agent must be able to answer queries from the user as to why it is performing a given operation, especially when part of a complex activity;
- providing grounding context: since the user may lose attention or perform some other task, the system must be able to provide sufficient dialogue context on demand to allow easy reentry to the conversation;
- concurrency: the system must be able to discuss multiple, simultaneous tasks in a coherent and natural way.

We outline a Dialogue Management system we have been building at CSLI to address the above challenges, which are pertinent to the workshop. We focus particularly on the *Activity Tree*, the central component used by the dialogue manager to mediate the communication between the human operator and the autonomous device. The Activity Tree models the status of the activities being performed by the dialogue-enabled autonomous agent, providing a means for the dialogue manager to address the challenges above.

The system has been extensively developed with respect to the command and control of a (simulated) helicopter UAV (Lemon, Gruenstein, & Peters 2002; Lemon *et al.* 2002); in this application, the UAV is given potentially complex tasks to perform, including multiple concurrent tasks. It has also been been used for applications that raise related issues which require attention to human-level detail, including intelligent tutoring (Clark *et al.* 2001) and control of in-car devices. Distraction and inattention are particularly important issues for the in-car application, and interactions can be very drawn out with large gaps in between (*e.g.* when navigating a route).[1]

Outline of System Architecture

The CSLI dialogue system is designed around a component-based architecture, making use of existing systems for speech recognition (Nuance), Natural Language parsing and generation (Gemini), and speech synthesis (Festival and Nuance Vocalizer). These and other components (e.g. interactive map GUI) are integrated using an event-driven loosely coupled distributed architecture, allowing easy replacement of components by others of similar functionality. The components described in this paper were all implemented in Java.

Following successful speech recognition and parsing we obtain a *logical form (LF)*: i.e. a domain-independent logic-based representation of the utterance's content. Logical forms are typically also tentatively classified as a type of *dialogue move*: e.g. whether it is a command, a question, an answer to a question, etc. Logical forms are passed to the Dialogue Manager, which performs such processes as resolving pronouns and other references; resolving any am-

[1]The in-car project is a very new one, and we expect to have many new issues to report by the time of the workshop.

biguities, or generating appropriate questions to do so; reclassifying the tentative dialogue move based on dialogue context; taking the appropriate action corresponding to the incoming dialogue move (e.g. a *command* may activate a new agent task); and generating any appropriate response.

One of the core components of the Dialogue Manager is the Dialogue Move Tree (DMT). Any ongoing dialogue constructs a particular DMT representing the current state of the conversation, whose nodes are instances of the dialogue move types. The DMT is used to interpret how an incoming LF relates to the current dialogue context: any new node attaches to an *active* node in the DMT as appropriate. For example, an answer will attach to its corresponding question node; an acknowledgment to a command. New conversation topics spawn new branches, giving the tree structure. The DMT thus acts as a history of dialogue contributions, organized by topic or "thread" based on activity. Since we are specifically focused on conversation about tasks and activities, each topic is associated with a task; hence, all significant DMT nodes[2] are linked to an activity node—i.e. a node in the Activity Tree (discussed below).

The multi-threaded nature of a DMT is an important way in which it differs from similar concepts (e.g. (Ahrenberg, Jonsson, & Dalhbeck 1990)). In particular, since *all* threads of a dialogue are represented and can be active simultaneously, a new utterance can be flexibly interpreted, even when it is not directly related to the current thread (e.g. a user can ignore a system question and give a new command, or ask their own question). This enhances the power of the dialogue interface for controlling autonomous agents in unpredictable dynamic environments.

A more complete description of the Dialogue Manager and the DMT can be found in (Lemon, Gruenstein, & Peters 2002; Lemon *et al.* 2002). In this paper, we focus on the role of the *Activity Tree* in addressing the dialogue challenges enumerated above.

Dialogue History and the Activity Tree

One of our aims for task-oriented dialogue is for the user to be able to send the device commands to be executed and then monitor the status of the corresponding activities as they are executed. Moreover, in the case of some devices, certain commands may result in the need for *joint-activities* which require collaboration between the human operator and the device. As the human operator works with the device, it is critical that the human and device maintain a shared conception of the state of the world, and in particular the status of each activity being performed; otherwise, serious problems related to *mode confusion* can result (Bachelder & Leveson 2001). This is especially important in interactions which take place over extended periods of time.

The Dialogue Manager incorporates a rich *Activity Model* in order to mediate between it and the autonomous behavioral agent. The Activity Model comprises a language for writing activity scripts (described below) which support conversation about the activities the agent actually performs.

[2]"Insignificant" DMT nodes are for utterances not about any topic, such as "pardon".

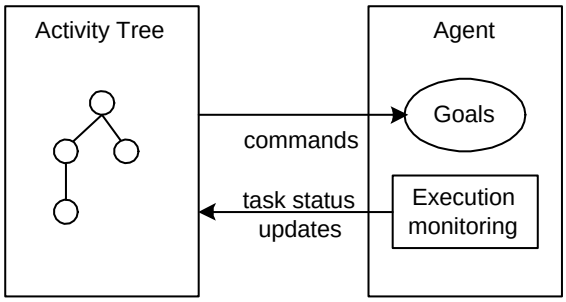

Figure 1: Interaction between Activity Tree and device

These activity representations play a number of important roles within the dialogue manager, such as: identifying the properties of activities that must be specified by further dialogue; disambiguating requests; and providing a framework for negotiating resource-conflict. However, the main use we focus on is their use in the *Activity Tree* to model the status of activity-execution.

The dialogue manager uses the *Activity Tree* to manage the (hierarchically decomposed) activities relevant to the current dialogue. This representation does not have to correspond exactly to the tasks actually being performed by the device—the Activity Tree represents tasks to the level at which dialogue about them is appropriate. In particular, low-level detailed tasks which the agent actually executes may not be appropriate for reporting to or discussing with the human, and so need not be represented as activities on the Activity Tree.

In general, when the human operator utters a command, it is parsed and then translated into a partially specified *activity*, which is then added to the Activity Tree. The system attempts to fully *resolve* the activity—e.g. resolving pertinent noun phrases for example, or asking the human operator questions in order to elicit further needed information. Once the activity has been fully resolved—i.e., once enough information has been gathered so the activity can be executed—it is sent to the device to be planned and executed. As it is executed, the device updates the state of the activity, which is reflected in the activity's node on the Activity Tree, and the dialogue manager in turn is notified of these updates: see Figure 1.[3]

At any point, an activity can be in one of the following states: `not_resolved` (partially described), `resolved` (fully described), `request_send` (should be planned), `planned`, `sent`, `current`, `suspended`, `cancelled`, `done`, `skipped`, `failed_preconditions`, `constraint_violation`, `conflicts` (resource conflict with another activity). The Activity Tree, then, mediates all communication between the dialogue manager and the device, providing a layer of abstraction by which the dialogue manager can communicate with any task-oriented device. Figure 2 shows a snapshot of an Activity Tree

[3]Ideally, this process keeps the Activity Tree synchronized with the device's own task execution, although some lag is possible.

```
root
.t1(fight_fire (at school)) [current]
..t2(transport (water from lake to school)) [current]
...t3(pickup_at_location (water from lake)) [done]
....t4(go (to lake)) [done]
.....t5(take_off) [done]
.....t6(fly_atom (to lake)) [done]
....t7(pickup_object (water)) [done]
...t8(deliver (water to school)) [current]
....t9(go (to school)) [current]
.....t10(take_off) [skipped]
.....t11(fly_atom (to school)) [current]
```

Figure 2: Snapshot of an Activity Tree during the execution of a fight_fire event

during the execution of an activity in which a robotic helicopter is fighting a fire.

Such an architecture stands in contrast to one in which the dialogue manager simply sends off commands to the device and then forwards all reports send to it by the device to the human operator. First and foremost, the tree structure gives the dialogue manage an explicit means of representing the relationship among the activities being executed by the system. As will be discussed below, the decomposition allows the dialogue system to make intelligent decisions about what changes in the world are conversationally appropriate, and which need not be mentioned.

The dialogue manager itself can handle decomposition of simple tasks, but in general the expectation is that task decomposition and execution is handled by some external plan-execution engine residing on the device. This requires some integration with the plan-executor itself—notifications of task status changes need to be sent to the Activity Tree.[4] We have performed basic integration with NASA's Apex system (Freed 1998) and the JACK agent system (Howden *et al.* 2001). We are currently developing libraries of tools to make such integration smoother.

The Recipe Scripting Language

While the Activity Tree represents the relationship among activities, each activity constituting a node on the tree is an instantiation of a *recipe* (c.f. plan) from the system's *recipe library* for a particular device. Conceptually, this mirrors the proposals in (Pollack 1990); and it is similar in concept to the plan libraries in (Allen *et al.* 1995; Fitzgerald & Firby 1998; Rich, Sidner, & Lesh 2001). The recipes in the library, as well as particular properties of the library itself, are compiled from a *recipe script*, which must be written for the autonomous device that is to be controlled via the dialogue manager. The recipe script defines recipes for undertaking particular activities (often in the pursuit of particular goals). Recipes model the domain-dependent common-sense knowledge which is needed to give rise to the structures on the Activity Tree which the dialogue manager uses for interpreting and producing relevant utterances.

While the similarity to plans is obvious, it should be stressed that recipes are *not* used as plans since they are not executed: their purpose is to support conversation about the device activity. Each recipe requires the author to fill in special constructions which are irrelevant to the execution of the recipe, but critical to both *how* and *when* the dialogue manager will report changes in the states of activities. Recipes consist of the following components:[5]

- a set of typed **slots**, which represent the pieces of information needed before a recipe can be instantiated into an actual activity (or *plan*) capable of being executed by the device (or human operator), or of being expanded into further sub-activities;
- a **body**, specified in a formalism similar to that of PRS-LITE (Myers 1996), which operates over the set of slots that specifies how the activity should be decomposed further to accomplish its goals.[6]
- **device information** about the conditions under which the recipe may be executed (*preconditions*), the results of the actions described by the recipe (*goals*), the *resources* needed to perform the actions described by the recipe (resource list), and *constraints* over the way in which the actions will be performed;
- **linguistic / dialogue information** about how to describe under various circumstances (or when to refrain from describing) the instantiated activity as it is being performed.

The plan execution engine that executes tasks needs to be able to handle the complexity inherent in recipe bodies, of course, but some of the recipe fields are strictly for use by the dialogue manager: e.g. the linguistic/dialogue information, and also the device information.[7] The plans for actually controlling the device may be significantly more complex than those in the dialogue manager recipe scripts. In particular, atomic actions (*i.e.* those that cannot be decomposed) need not necessarily correspond to atomic actions of the device. The Activity Model recipes are simply written to match the device plans *to the level of detail to which dialogue about those plans and tasks are supported.* Hence, the Activity Model provides an important layer of abstraction from the complexity of the device, to the level at which it is appropriate to discuss tasks with the human operator.[8]

The Dialogue Manager / Activity Tree Interface

The utility of the Activity Model becomes apparent when it is examined in light of the interface to the device they

[4](Fitzgerald & Firby 1998) describes an architecture where the task modeling component is fundamentally integrated into the dialogue manager, controlling dialogue processes themselves as well as external tasks.

[5]A much more detailed description of the recipe scripting language can be found in Section 5 of (Gruenstein 2002).

[6]Note: the body is only used for execution if the dialogue manager itself is performing task decomposition.

[7]Some or all of the device information is likely to be reproduced in the device plans.

[8]Ideally, we would like to be able to simply compile appropriately annotated device-tasks, written in Apex, JACK, or some other plan language, directly into Activity Model recipes. For now, however, the recipes are hand-constructed.

provide for the dialogue manager. Given both the recipe library and the dynamically evolving activity tree, the dialogue manager is afforded information that allows it to communicate more effectively with the human operator about the state of the device. In this section, we discuss how this structure is leveraged in ways that is specifically important to the issue of extended interaction.

Task monitoring and reporting First and foremost, changes in the state of activities on the activity tree can give rise to communicative acts on the part of the dialogue system. In particular, as the system notices changes in the state of activities (for instance from `current` to `done`), the system may choose to inform the user of this change of state. For example, in Figure 2, when the `pickup(water)` task changes to `done` and the `deliver(water to school)` task becomes current, the following utterances are generated:

Sys: I have picked up water from the lake.
Sys: I am now delivering the water to the school.

The content of the task report is determined by the *linguistic information* specified in the recipe script that defines the activity that underwent a state change. In particular, included in that information is a section in which the library's designer can specify whether or not a change to a particular state should be announced by the system for that activity.

In addition, the dialogue system makes use of the activity tree to filter out reports about state changes which are no longer true at the time they are reported. For instance, if an activity changes state to `current` and then quickly changes state again to `done` before the dialogue manager has a chance to report that it has become *current*, this message is removed from the output agenda without being uttered.

Message content / level of detail When the system does decide to make a report on a state change of an activity, it must also decide on the content of this utterance. Depending on the state, a different level of detail is often desired. For example, when reporting on an ongoing activity (e.g. flying to a destination), properties of the activity may be relevant (e.g. speed and altitude), whereas the same properties need not be mentioned when the completion of the task is reported (e.g. "I have flown to the tower").

Information on which properties to report when can be encoded into the recipe scripts. For example, the recipe for `fly` is annotated so that the values of the `toLocation`, `toAltitude`, and `toSpeed` slots are reported when the activity is `current`; but that only the `toLocation` is reported when the activity is `done` or `suspended`.

Explanation: answering *Why* questions Due to the complexity of some activities and the length of time it takes to perform a particular activity, it may not always be immediately apparent to the operator why the device is performing a particular action. The operator may simply have forgotten that he or she gave a particular command, or perhaps may not realize that the system is doing a particular activity because the activity is actually a sub-activity of another. For example, the UAV application contains a relatively complex activity called *fight_fire* in which the helicopter repeatedly picks up water at one location, transports the water to a second location where a building is on fire, and drops loads of water there until the fire has been extinguished. Because this activity is relatively complex and has a long duration, it's possible that the operator might want to question the helicopter as to why it is, say, flying to the lake.

The Activity Tree is used by the dialogue manager to provide a means of answering such *why* questions. In order to answer why the device is performing a particular activity, the dialogue manager can look at the activity's ancestor nodes on the Activity Tree and simply report an appropriate ancestor. Referring again to the snapshot of the Activity Tree as it might appear during one stage of fighting the fire at the school in Figure (2), we note that at this moment in time the helicopter has picked up the water and is carrying it to the school to extinguish the flames.

Given this Activity Tree, the Dialogue Manager supports such queries as the following:

- Why?
- Why did you pick up the water at the lake / go to the lake / take off / pick up the water?
- Why are you delivering the water to the school / going to the school?

In order to answer each of these questions, the dialogue manager must first determine which activity specifically the user is asking a *why* question about. Once this has been determined, it must choose the appropriate ancestor of this activity to report as an answer to the question. Most of the time, this is simply the parent of the activity in question.[9]

In order to answer *why* questions such as those above, the dialogue manager uses the algorithm in Figure 3. Note that the input to the algorithm is a logical form representing a *why_query*. It is assumed that the format of the logical form is the following:

```
why_query(ActivityMarker,
          ActivityDescription)
```

where `ActivityMarker` can have one of the following values:

- `anap`: for the purely anaphoric utterance of *why?*
- `currActivity`: for utterances referring to the current activity, either *Why are you Xing?* or *Why are you doing that?*
- `complActivity`: for utterances referring to a completed activity, either *Why did you X* or Why did you do that?

[9] There is one case, however, in which the parent is not an appropriate response – namely, the case in which the natural language description of the parent activity is identical to that of the child. For instance, in the UAV library there are two recipes having to do with the activity of *flying*: `fly` and `fly_atom`, where the first can optionally include taking off and the second includes only the act of flying from one waypoint to another. Naturally, `fly_atom` often appears as a subtask of `fly`, even though the natural language report generated to describe both of these activities will be identical.

```
Algorithm:  ANSWER WHY QUERY
Given:  The logical form w of a why_query
a = find_relevant_activity(w)
r_a = generate_logical_form(a)
p = parent(a)

while(p != null) {
   r_p = generate_logical_form(p)
   if(r_p != r_a) return why_answer(r_p)
   p = parent(p)
}
Algorithm:  FIND RELEVANT ACTIVITY
Given:  logical form w of a why_query
        with ActivityMarker m and
        ActivityDescription d
Given:  list of salient activities S
if m = anap AND d = anap
   return first(S)
foreach s in S {
   if (s matches a in state and description)
        return s
}
```

Figure 3: Explanation algorithm

and `ActivityDescription` has either the value of `anap` for utterances that don't refer to a specific activity (e.g. *Why?* and *Why are you doing that?*), or the logical form for the command that triggered the activity.

Providing context for conversation reentry Task-oriented conversations over extended periods raise the possibility of a break in the interaction, for various possible reasons: e.g. a long task (such as flying to a waypoint) without communication; or the user being distracted by some other task and lose attention. The issue of interrupted conversation was first raised by our in-car application, to which it is particularly important: conversation needs to be terminated if the driver becomes distracted or needs to focus on the driving task at a given moment.

On reentering a conversation, especially if after more than a few seconds or after working on a different task, the user may require a reminder of the topic of discussion at the break point. A more complete setting of context may involve including a precis of the wider topic of conversation—i.e. a higher-level task which the under-specified task is part of. For example, in the car domain we may be interrupted while choosing a restaurant:

User: What were we talking about?
Sys: We were selecting a restaurant. I asked you which type of cuisine you wanted.

This is not the same as the dialogue-summarization problem (e.g. (Zechner 2001)). We do not need to provide a full summarization of even a snippet of dialogue, which may meander between multiple topics—e.g. in between setting the mission-level task and requesting the flight to the house, there may have been sub-dialogues or dialogue about other topics. In our case, an appropriate summary consists of context pertinent to the point at which the dialogue was interrupted.

```
Algorithm:  GENERATE CONTEXT DESCRIPTION
context = new Stack()
n = most_recent_DMT_node
context.push(n)
if n is a closed node then {
   n = parent(n)
   r_n = extract_logical_form(n)
   context.push(n)
}

a = find_associated_activity(n)
while (a is not root) {
   r_a = generate_logical_form(a)
   context.push(r_a)
   a = parent(a)
}

generate_NL_description(context)
```

Figure 4: Context description algorithm

The required context description can be constructed using the Dialogue Move Tree as a dialogue history, and the Activity Tree as a history of task execution. The most recent node n in the DMT represents the most recent utterance. n also contains a link into the Activity Tree to the corresponding task under discussion, which contains a link back up to its parent task (if it is a sub-task), etc. This can be traced all the way back up to the root of the Activity Tree, which corresponds to the most abstract description of the current topic.

For example, suppose the restaurant-search dialogue has the following active node n:

```
wh_question(which(cuisine))
```

and that the Activity Tree contains the following task nodes:

```
.task1(select_restaurant)  [current]
..task2(choose_neighborhood)  [done]
..task3(choose_cuisine)  [current]
```

with DMT node n containing a link to `task3`. A description of n is pushed onto a stack,[10] followed by status reports for the task associated with n—`task3`—and its parent—`task1`. If n is a *closed* node—e.g. an answer to a question—then we also push its parent DMT node (i.e. the description contains both the question and then given answer).

An algorithm is given in Figure 4.

Note that the description generated for most recent active node n depends on the type of dialogue move n corresponds to. The examples above correspond to question-type moves. For a command issued by the User, the execution

[10] We use a stack since the context description is built in reverse historical order.

status of the corresponding task may have changed (depending on how long the conversation was suspended). Since the Activity Tree maintains the status of tasks, this can be incorporated into the description. For example, given DMT node

```
command(fly, to(tower1))
```

and corresponding Activity Tree node

```
task1(fly(from base to tower
      speed high ...))       [done]
```

the system would generate a summary such as the following:

User: What were we talking about?
Sys: You asked me to fly to the tower. I have completed that task / I have flown there.

We expect this general technique to be effective across all our application domains. For example, in the tutoring domain, activity nodes correspond to lesson-topics, which should lead to summaries that allow students to pick up after suspending a tutoring session:

User: What were we doing?
Sys: We were reviewing your performance in handling a fire in the hold.

Conclusions and Further Work

Dialogue, and particularly the requirement on the explicit representation of context and interaction history, provides a useful mechanism for addressing some of the important issues for interaction about activities that occur in complex domains over an extended period.

We have described how the use of an explicit Activity Model in our Dialogue Manager provides a framework against which to build a number of techniques which are useful in extended interactions: task reporting; report-content management; explanations of activities; and context description for interrupted conversations.

As future work, we are constantly improving on the form of language generated in the above tasks, and plan to address dynamic revision of level-of-detail. Extended interaction also introduces the possibility of dynamic registration of new devices, which requires both functionality advertising (i.e. dynamically extending an activity model)—the easy part!—as well as dynamically extending linguistic and dialogue capabilities (Rayner *et al.* 2001).

References

Ahrenberg, L.; Jonsson, A.; and Dalhbeck, N. 1990. Discourse representation and discourse management for natural language interfaces. In *In Proceedings of the Second Nordic Conference on Text Comprehension in Man and machine*.

Allen, J. F.; Schubert, L. K.; Ferguson, G.; Heeman, P.; Hwang, C. H.; Kato, T.; Light, M.; Martin, N. G.; Miller, B. W.; Poesio, M.; and Traum, D. R. 1995. The TRAINS project: A case study in building a conversational planning agent. *Journal of Experimental and Theoretical AI* 7:7–48.

Allen, J.; Byron, D.; Dzikovska, M.; Ferguson, G.; Galescu, L.; and Stent, A. 2001. Toward conversational human-computer interaction. *AI Magazine* 22(4):27–37.

Bachelder, E., and Leveson, N. 2001. Describing and probing complex system behavior: A graphical approach. In *Proceedings of the Aviation Safety Conference*.

Clark, B.; Fry, J.; Ginzton, M.; Peters, S.; Pon-Barry, H.; and Thomsen-Gray, Z. 2001. Automated tutoring dialogues for training in shipboard damage control. In *Proceedings of SIGdial 2001*.

Clark, H. H. 1996. *Using Language*. Cambridge University Press.

Fitzgerald, W., and Firby, R. J. 1998. The dynamic predictive memory architecture: integrating language with task execution. In *IEEE Symposium on Intelligence and Systems*.

Freed, M. A. 1998. Managing multiple tasks in complex, dynamic environments. In *AAAI'98*.

Gruenstein, A. 2002. Conversational interfaces: A domain-independent architecture for task-oriented dialogues. Master's thesis, Stanford University.

Howden, N.; Ronnquist, R.; Hodgson, A.; and Lucas, A. 2001. JACK intelligent agents—summary of an agent infrastructure. In *Int'l Conf. on Autonomous Agents*.

Lemon, O.; Gruenstein, A.; Battle, A.; and Peters, S. 2002. Multi-tasking and collaborative activities in dialogue systems. In *Proceedings of 3rd SIGdial Workshop on Discourse and Dialogue*, 113 – 124.

Lemon, O.; Gruenstein, A.; and Peters, S. 2002. Collaborative activities and multi-tasking in dialogue systems. *Traitement Automatique des Langues (TAL)* 43(2):131 – 154. Special Issue on Dialogue.

Myers, K. 1996. A procedural knowledge approach to task-level control. In *3rd International Conference on AI Planning Systems (AIPS-96)*.

Pollack, M. E. 1990. Plans as complex mental attitudes. In Cohen, P. R.; Morgan, J.; and Pollack, M. E., eds., *Intentions In Communication*. Cambridge, MA: MIT Press. chapter 5, 77–103.

Rayner, M.; Lewin, I.; Gorrell, G.; and Boye, J. 2001. Plug and play speech understanding. In *2nd SIGdial Workshop on Discourse and Dialogue*.

Rich, C.; Sidner, C. L.; and Lesh, N. 2001. COLLAGEN: Applying collaborative discourse theory to human-computer interaction. *AI Magazine, Special Issue on Intelligent User Interfaces*.

Zechner, K. 2001. Automatic generation of concise summaries of spoken dialogues in unrestricted domains. In *SIGIR'01*.