RavenClaw: Dialog Management Using Hierarchical Task Decomposition and an Expectation Agenda

Dan Bohus *Alexander I. Rudnicky*

Carnegie Mellon University,
Pittsburgh, PA, 15213
{dbohus,air}@cs.cmu.edu

Abstract

We describe RavenClaw, a new dialog management framework developed as a successor to the Agenda [1] architecture used in the CMU Communicator. RavenClaw introduces a clear separation between task and discourse behavior specification, and allows rapid development of dialog management components for spoken dialog systems operating in complex, goal-oriented domains. The system development effort is focused entirely on the specification of the dialog task, while a rich set of domain-independent conversational behaviors are transparently generated by the dialog engine. To date, RavenClaw has been applied to five different domains allowing us to draw some preliminary conclusions as to the generality of the approach. We briefly describe our experience in developing these systems.

1. Introduction

Dialog management maintains continuity over turns in a conversation between human and computer. While many approaches have been developed to address this problem, we believe that the essence of dialog management resides in performing two functions: interpreting user inputs with respect to task(s) within the domain, and maintaining the coherence, over time, of the conversation.

Task structure is not always explicitly represented in dialog systems. For example, in graph-based systems ("IVR" systems) task structure is implicit in the structure of the graph. In information access systems, the "task" consists of form-filling and is again implicitly represented in the architecture. Explicit task representations, however, are necessary for more complex domains, for example travel planning. The CMU Agenda dialog manager [1] represents one approach to directly modeling the human's task. The current RavenClaw dialog manager builds on the experience of Agenda, notably in providing a clear separation between task-specific behavior and more general discourse behaviors (which we refer to as "conversational strategies"). The system development and maintenance effort is entirely focused on providing a description of the task to be performed; the mechanisms for maintaining the coherence and continuity of the conversation are generated by an underlying dialog engine.

Section 2 describes in detail the structure, mechanisms and functionality of the RavenClaw architecture. Subsequently, Section 3 summarizes our experience in using it in five different applications that span a variety of task types. Finally, Section 4 concludes the paper and presents our current and future plans for extending this framework.

2. RavenClaw architecture

RavenClaw is a two-tier architecture (Figure 1). The ***Dialog Task Specification*** layer captures all the domain-specific dialog logic. The ***Dialog Engine*** is a domain-independent component that controls the dialog by executing the Dialog Task Specification, and contributes basic conversational strategies (e.g., timing and turn-taking behavior, grounding behavior; universal dialog mechanisms like help, repeat, suspend/resume). All domain-specific information is clustered within the Dialog Task Specification level. System developers can therefore focus their attention on defining the domain-specific control of the dialog, and delegate realization of generic dialog mechanisms to the Dialog Engine.

2.1. The Dialog Task Specification

The domain-specific dialog control is represented in the Dialog Task Specification level using a tree of dialog agents, with each agent handling a certain part of the dialog task.

To exemplify, Figure 1 illustrates the top portion of the dialog task tree for RoomLine, a spoken dialog system for conference room reservation and scheduling. The root node subsumes several children: Login, which identifies the user to the system, GetQuery, which obtains the time and room constraints from the user, GetResults, which executes the query against the backend, and DiscussResults which presents the obtained results and handles the forthcoming negotiation for selecting the conference room that best matches the user's needs. Moving one level deeper in the tree, Login decomposes into Welcome, which introduces the user to the system, AskRegistered and AskName, which identify the user, and finally GreetUser, which sends a greeting to the user.

Hierarchical task decompositions, traditionally used for task execution in robotics, have gained popularity in the dialog management community. Examples include the use of a tree-of-handlers in Agenda Communicator [1], activity trees in WITAS [2] and recipes in Collagen [3]. The hierarchical representation has several advantages. Most (goal-oriented) dialog tasks have an identifiable structure which naturally lends itself to a hierarchical description. The subcomponents are typically independent, leading to ease in design and maintenance, as well as good scalability properties. Moreover, the tree structure can be easily extended at run-time, allowing for the dynamic construction of dialog structure, a very useful feature in certain types of tasks. Finally, the tree representation implicitly captures the notion of context (via the parent relationship), as well as a default chronological ordering of the actions (i.e. left-to-right traversal); these elements significantly simplify the design of a dialog rendering engine operating over this type of representation.

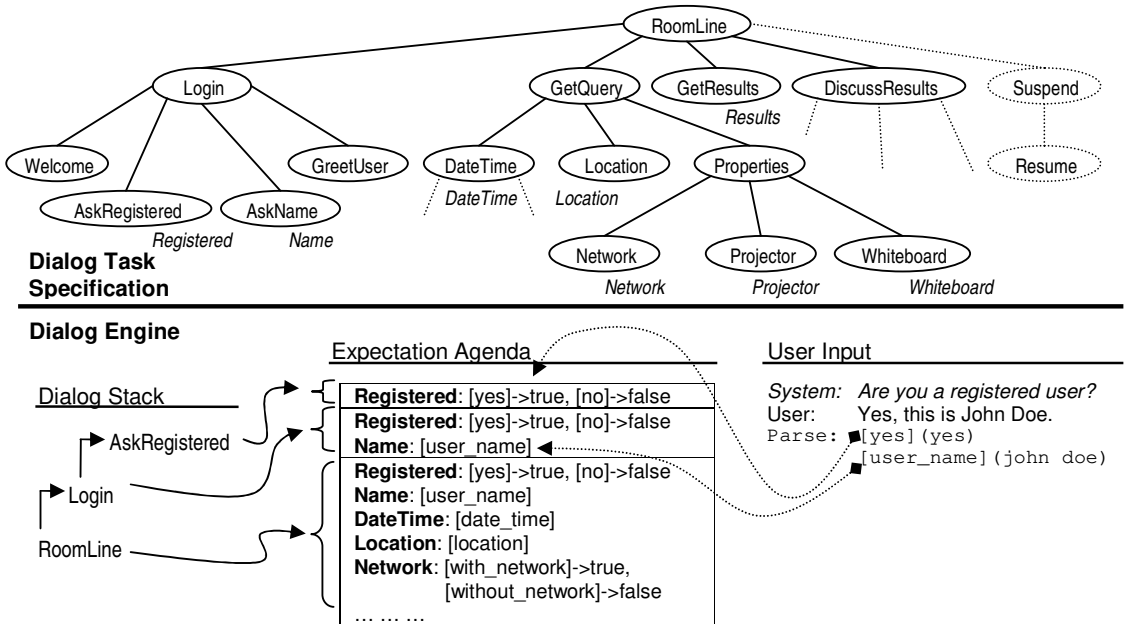

Figure 1: RavenClaw architectural details

As a prerequisite to a more detailed presentation of the Dialog Engine control mechanisms (Section 2.2), we first turn our attention to the structure and functionality of the dialog task agents.

2.1.1. Dialog Task agents

Two categories of dialog agents populate the task tree: ***fundamental dialog agents*** and ***dialog agencies***.

The fundamental dialog agents appear as leaf nodes (i.e. Welcome, AskRegistered) and represent atomic dialog actions. RavenClaw uses four types of fundamental agents: ***Inform*** - sends an output (e.g. Welcome), ***Request*** - requests information (e.g. AskRegistered), ***Expect*** - expects information, but without requesting it (e.g. Projector) and ***DomainOperation*** - performs other domain-related operations (e.g. GetResults). The non-terminal nodes in the tree are ***dialog agencies*** (e.g. Login, GetQuery); agencies control the execution of their subsumed agents, capturing the higher level temporal and logical structure of the dialog task.

Each agent implements an Execute routine, and holds a set of preconditions and triggers, and a completion criterion. The Execute routine is specific to the agent type. For example, Inform-type agents simply generate an output when executed, while Request-type agents also trigger an Input Phase (see subsection 2.2.2) to collect the user's response. For agencies, the Execute routine is in charge of planning the order of the execution of the sub-agents. This sub-task planning problem is currently resolved by combining a set of simple policies (i.e. left-to-right traversal), with the preconditions that each agent holds. The system is however open to more sophisticated policies, and even learning at the dialog task level (e.g. by casting the sub-agent planning problem as a Markov Decision Process [4]).

Between the preconditions, triggers, completion criteria and the Execute routines the tree captures an overall hierarchical plan for the dialog task but does not prescribe a fixed order of execution (as might be found in a directed dialog system). When executed, a particular trace through this plan is generated based on the specified policies, encoded domain constraints and logic, as well as the user's inputs.

An important feature of dialog agents, qualifying them as more than plan operators, is their ability to store ***concepts***, and participate in the Input Phase, in which the information collected from the user is incorporated into the system. Each agent can contain one or more concepts (e.g. *Registered*, *Name*) that hold task-related information. Concepts are represented as probability distributions over the set of possible values, enabling a grounding management layer based on belief updating and decision making under uncertainty.

2.2. The Dialog Engine

The Dialog Engine is the core component in RavenClaw and controls the dialog by executing the Dialog Task Specification. Dialog flow is generated by interleaving ***Execution Phases*** and ***Input Phases***. In an Execution Phase, the various agents in the task tree are executed and generate the system's behavior. In an Input Phase, the system collects and incorporates the information from the user's input. We now describe these mechanisms is more detail.

2.2.1. The Execution Phase

The Dialog Engine uses a stack to track the dialog structure and schedule the agents in the task tree for execution (see Figure 1). Initially, the root agent is placed on the dialog stack. Subsequently, the engine repeatedly takes the agent currently on the top of the stack, and executes it. When agencies are executed, they typically schedule one of their descendants for execution by pushing it on the dialog stack. Ultimately, the execution of fundamental dialog agents generates the system's responses and actions.

Note that the isomorphism between the dialog stack and the dialog tree is only apparent. There is an essential functional difference between the two structures: the stack captures the temporal and hierarchical structure of the current dialog, while the tree describes the dialog task, implicitly capturing the set of all possible dialogs in the domain. As described in the next subsection, the user can at any point take the initiative and shift the focus of the conversation to another part of the task (as long as the domain logic and constraints encoded in the task tree are not violated). This can lead to the introduction of new dialog agents on the stack, breaking the apparent isomorphism. For instance, if the user had responded by saying *"Suspend"* to the system's *"Are you a registered user?"* question, the Suspend agency would be triggered and placed on the stack on top of the AskRegistered agent. Moreover, the Dialog Engine itself can push new agencies modeling various conversational strategies on the dialog stack (see Section 2.3). The stack therefore tracks the current structure of the dialog, and provides support for focus shifts and handling sub-dialogs, as well as for the construction of the system's agenda of expectations.

The Request-type fundamental agents can interrupt the Execution Phase and instruct the Dialog Engine to start an Input Phase. The engine then acquires and incorporates the input from the user, as described below.

2.2.2. The Input Phase

Each Input Phase consists of three stages: (1) constructing the agenda of expectations, (2) binding values from the input to concepts, and (3) analyzing the need for a focus shift.

In the first stage, the system constructs the expectation agenda, a data-structure describing what the system is expecting to "hear" at this point. The agenda is constructed by traversing the dialog stack in a top-down manner and asking each of the agents encountered to declare their expectations. An ***expectation*** describes the semantic grammar slots an agent is looking for (e.g. [user_name] for AskName in Figure 1), which concept they update (e.g. *Name*), and how the update is to be performed. An agency's expectation is defined by collecting the expectations of all of its descendants. The resulting expectation agenda will therefore contain multiple sections (see Figure 1) representing increasingly larger contexts, imposed by the current state of the dialog stack.

In the second stage, information from the input is matched to the declared expectations by a top-down traversal of the agenda. The top-down traversal provides support for reference resolution: if expectations for the same grammar slot exist in different sections of the agenda, the ones that are placed higher (and therefore closer in context to the conversational focus) will take precedence. In the example from Figure 1, the [yes] slot is bound to the Registered concept (setting its value to `true`), but also [user_name] is bound to the *Name* concept. When the execution later resumes, the AskName agent already has its completion criterion satisfied (i.e. the *Name* concept is available), and will not be scheduled for execution.

Finally, in the last stage of the Input Phase, the system establishes if any of the dialog agents in the task tree need to be brought into focus, in light of the recently gathered information. This process is similar to the construction of the expectation agenda, in that each of the agents in the task tree is given the opportunity to declare a focus claim. Focus claims are domain-dependent, and they are specified as trigger conditions on the agents. If the need for a focus shift is signaled, the claiming agent is pushed on the dialog stack. The Input Phase concludes, and a new Execution Phase begins with the agent on top of the stack.

2.3. Conversational strategies

A characteristic which greatly influences the usability and ultimately the success of spoken dialog systems is their ability to employ a rich set of conversational strategies. These encompass grounding behaviors (e.g. confirmations, disambiguations, channel reestablishment, etc) turn-taking and timing behaviors, as well as other generic dialog mechanisms, like the ability to handle requests for help, for repeating the last utterance, suspending and resuming the dialog, starting over, re-establishing the context.

RavenClaw provides automatic support for all the above-mentioned conversational strategies. Internally, they are implemented as dialog agencies, in the same manner as the domain-specific dialog task tree. Their corresponding sub-trees are either inserted in the task tree at the beginning of the session (e.g. the Suspend agency handling requests for temporarily suspending the dialog in Figure 1), or are pushed by the Dialog Engine onto the dialog stack at an appropriate time (e.g. explicit confirmations, disambiguations, etc). The transparent support offered for conversational strategies considerably simplifies the system development and maintenance efforts, while also providing uniformity within and across systems. Finally, if need be, the developers can build new task-specific conversational strategies, and/or overwrite the available defaults.

3. RavenClaw-based systems

Evaluating the appropriateness of a dialog management framework is a challenge, as many applications can be (and are) recast into a form that is tractable within a particular approach. As a first step, we made the system available to different developer teams working in a variety of structurally different domains and noted the degree of accommodation required by RavenClaw. Below, we briefly comment on the development of five such systems using RavenClaw-based dialog managers (Table 1 lists some system characteristics).

LARRI [5] is a multi-modal conversational agent which provides assistance to F/A-18 aircraft mechanics performing maintenance tasks. The system guides the user through the task at hand, and provides access to relevant stored information (e.g. diagrams, warnings, etc). The tree-based decomposition of the dialog task is well suited to this domain, as it maps directly onto the structure of the actual tasks to be performed. Moreover, since the tasks are extracted on-the-fly from a task repository, the framework's ability to generate/expand the dialog task tree at runtime plays an important role. The number of agents in the task tree can grow from 61 to several hundreds, depending on length of the task to be performed; in practice, the system scales gracefully. LARRI was used to initially test the RavenClaw design.

The **Intelligent Procedure Assistant** [6], developed at NASA/Ames operates in a very similar domain: the system is intended to provide assistance to astronauts on the International Space Station in the execution of procedural tasks and checklists. In contrast to the other systems in this section, which use Phoenix [7] for semantic parsing and are

integrated in a Galaxy architecture [8], the IPA uses the Gemini [9] parser and language generator, and is integrated using Open Agent Architecture [10]. Although new input and output representations reflecting the Gemini semantics were added, RavenClaw's modular design allowed us to adapt it to this new setting without any core structural changes.

BusLine, developed as part of the "Let's Go!" project [11], provides an information search interface to Pittsburgh bus schedules. In contrast to LARRI and IPA, BusLine uses a static dialog task tree. BusLine required several changes to RavenClaw to better support information exploration: maintaining a history of previous values for the concepts in the task tree, providing clearer semantics for re-opening topics (agencies) for conversation, as well as refining the default behaviors of some conversational strategies.

RoomLine provides assistance for conference room reservation and scheduling within the School of Computer Science at CMU. The system's task entails both information access and room-schedule modification functionality. The RavenClaw architecture supports the RoomLine functionality without any modification.

TeamTalk provides spoken control of a team of robots, and focuses on managing multi-way conversations and the asynchronous behavior that characterizes a team of autonomous agents. The initial design uses a separate dialog manager instantiation for each robot, and a token-passing scheme to control turn-taking. Although multi-participant conversation is a novel application, no structural changes to RavenClaw appear to be necessary.

System	Domain Type	Interaction Type	# of agents	# of concepts
LARRI	Guidance & Browsing	System Guided	61 +	31 +
IPA	Guidance & Browsing	System Guided	52 +	25 +
Bus Line	Information Exploration	Mixed Initiative	44	10
Room Line	Information Mgmt.	Mixed Initiative	50	9
Team Talk	Command & Control	User Initiative	~80 estim.	~20 estim.

Table 1: Five RavenClaw-based dialog systems

4. Conclusions

We described RavenClaw, a new dialog management framework for spoken dialog systems operating in complex, goal-oriented domains. The framework separates the domain-dependent and domain-independent components of the dialog manager and focuses system development effort on defining a hierarchical decomposition of the underlying task. A Dialog Engine uses the task representation to drive the dialog forward towards its goals, and uses separate, generic, conversational strategies to maintain dialog coherence and continuity.

RavenClaw-based dialog managers were constructed for five dialog systems spanning qualitatively and quantitatively different domains. Work in one domain (information exploration) resulted in the addition of new functionality but no major changes in the overall structure or core mechanisms were required. Moreover, the framework easily adapted to all these domains, indicating a high degree of versatility and scalability.

Currently our efforts are focused on managing grounding in a separate layer based on the continuous updating of the system's beliefs about the validity of information in the task tree, and decision making under uncertainty.

5. Acknowledgements

We would like to thank Antoine Raux, Brian Langler, June Sison, Thomas Harris, Satanjaneev Banerjee, S.P. Kishore, the primary developers of the BusLine and TeamTalk systems. We would also like to thank Ridy Lie for his work on the development of the RoomLine system, and the RIALIST group at NASA/Ames for their collaboration in introducing a RavenClaw-based dialog manager into IPA.

This research was sponsored in part by the Space and Naval Warfare Systems Center, San Diego, under Grant No. N66001-99-1-8905. The content of the information in this publication does not necessarily reflect the position or the policy of the US Government, and no official endorsement should be inferred.

6. References

[1] Xu, W. and Rudnicky, A., "Task-based Dialog Management Using an Agenda", ANLP/NAACL 2000 Workshop on Conversational Systems, May 2000.

[2] Lemon, O., Gruenstein, A., Battle, A., and Peters, S. "Multi-tasking and Collaborative Activities in Dialogue Systems", Proceedings of 3rd SIGDIAL Workshop on Discourse and Dialogue, Philadelphia, p.113-124, 2002.

[3] Rich, C., and Sidner, C., "Collagen: A Collaboration Agent for Software Interface Agents", *Journal of User Modeling and User-Adapted Interaction*, vol.8, 1998

[4] Litman, D. J., Kearns, M. S., Singh, S., and Walker, M. A., "Automatic Optimization of Dialogue Management", In Proceedings of COLING 2000.

[5] Bohus, D. and Rudnicky, A., "LARRI: A Language-Based Maintenance and Repair Assistant", IDS-2002, Kloster Irsee, Germany.

[6] Aist, G., Hockey, B.A., Rayner, M., Hieronymus, J., Bohus, D., Boven, B., Blaylock, N., Campana, E., Early, S., Gorrell, G., and Phan, S., "Talking Through Procedures: An Intelligent Space Station Procedure Assistant", EACL-Demo, 2003

[7] Ward, W., and Issar, S. "Recent Improvements in the CMU Spoken Language Understanding System", Proceedings of the ARPA HLT Workshop, March 1994

[8] Seneff, S., Hurley, E., Lau, R., Pao, C., Schmid, P., and Zue, V. "Galaxy-II: A reference architecture for conversational system development", Proceedings ICSLP, Sydney, Australia, 1998

[9] Dowding, J., Gawron, J.M., Appelt, D., Bear, J., Cherny, L., Moore, R., and Moran, D "A Natural Language System for Spoken-Language Understanding", Meeting of the Association for Computational Linguistics

[10] D. Martin and A. Cheyer and D. Moran, "The Open Agent Architecture: a framework for building distributed software systems", *Journal of Applied Artificial Intelligence,* vol 13, no. ½, pp 91-128, 1999

[11] Raux, A., Langner, B., Black, A., and Eskenazi, M., "LET'S GO: Improving Spoken Dialog Systems for the Elderly and Non-natives", submitted to EuroSpeech 2003